Journal of Electronic Testing (2023) 39:583–595
https://doi.org/10.1007/s10836-023-06087-2

Trade-off Mechanism Between Reliability and Performance for Data-flow Soft Error Detection

Zhenyu Zhao[1] · **Xin Chen**[1] · **Yufan Lu**[2]

Received: 21 April 2023 / Accepted: 5 October 2023 / Published online: 2 November 2023

Abstract
The high energy particles in the space environment will perturb integrated circuits, resulting in system errors or even failures, which is also known as single event effects (SEE). To ensure the normal operation of space systems, it is first necessary to detect these errors. However, detection algorithms also bring additional overhead to the system and reduce its performance. Therefore, we aim to find a trade-off between reliability and performance. To this end, we propose a quantitative evaluation model for detection methods that evaluates the reliability gain of different detection methods under the same overhead. Our method allocates the optimal detection method to the corresponding code segment based on the quantitative results, thereby achieving a trade-off between reliability and performance. Experimental results show that the average energy efficiency of our trade-off method is 91.34%, which is 21.49% higher than the other methods.

Keywords Single-event effects · Soft error · Fault-tolerant system · Reliability · Trade-off · Software-based detection techniques

1 Introduction

Radiation hardened technology are essential for integrated circuits working in space, the reason is that integrated circuits will be perturbed in space due to the hit of high energy particle, which is also called single event effects (SEE). One typical type of SEEs is single event upset (SEU), the phenomenon of SEU is the logic state of sequential logic flips from "0" to "1" (and vice versa), which may cause system failures [3, 7, 11, 18, 29].

Fortunately, SEU is one kind of soft error and can be recovered. The first step of recovery operation is detecting these soft errors. There is a large amount of literature on this classic topic [1, 2, 8, 10, 33]. However, detecting these soft errors will increase execution time, which means the degradation of system performance [9, 13, 21, 22, 24, 27, 31, 37]. For example, error detection by diverse data and duplicated instructions (EDDDDI) improves the reliability, but brings a huge overhead of execution time [24]. Translator for reliable software (ThOR) [30] implements code redundancy at C Code level. Because of the utilizing of high-level languages, this method is less difficult to implement. But due to the coarse-grained of high-level languages, the overhead of this method is higher than that of EDDDDI.

Therefore, implementing trade-off between reliability and performance has attracted more and more attentions. Reliable Code Compiler (RECCO) [6] and Partial Software Protection [42] implement the trade-off by only detecting the key variables and codes, such as variables or code segment that are used multiple times, but the key variables and codes are selected manually. Ref. [34] proposes a quantitative method to evaluate the detection method, however, this method does not consider the trade-off strategy between multiple detection methods.

Motived by this problem, a trade-off mechanism between reliability and performance for data-flow soft error detection is proposed in this paper. The main contributions of this paper are as follow:

Responsible Editor: A. Yan

✉ Xin Chen
xin_chen@nuaa.edu.cn

Zhenyu Zhao
15947470421@163.com

Yufan Lu
498425254@qq.com

[1] College of Electronic and Information Engineering, Nanjing University of Aeronautics and Astronautics, Beijing, China

[2] School of Computer Science and Electronic Engineering, University of Essex Colchester, London, UK

1. A mathematical model is developed to evaluate the detection methods. Based on this, a quantitative evaluation method is proposed to measure the detection energy efficiency of different types of detection methods.
2. We have implemented an automated test platform to find the optimal configuration of the proposed detection method according to the energy efficiency of different types of detection methods.
3. With the help of presented quantitative evaluation method and automated test platform, the optimal configuration of the detection methods can be obtained, and then the trade-off between reliability and performance can be realized more effectively.
4. The final results show that the average energy efficiency of our trade-off method is 91.34%, RECCO is 75.42%, partial software protection is 85.82%, EDDDDI is 66.34%, ThOR is 48.49%, and Fault Screening is 73.22%. This suggests that our trade-off method works better.

The remainder of this paper is organized as follows. Section 2 provides a brief overview of related works. Section 3 presents relevant mathematical models, and Section 4 presents the proposed trade-off strategy. Section 5 discusses experimental results, and conclusions are made in Section 6.

2 Related Works

Software-based soft error detection methods are divided into two types: program redundancy and assertion detection. Table 1 provides examples of two types of detection methods.

The fundamental concept of program redundancy detection methods is to detect errors by comparing the results of original code with the result of modified code. Currently, there are various program redundancy detection methods, such as software fault tolerance (SWIFT) [31], error detection through repeated instructions (EDDI) [23] and EDDDDI [24]. At the source code level, the detection methods include ThOR [30] and RECCO [4-6, 6].

Another low overhead soft error detection method is assertion detection. The fundamental concept of assertion detection is to extract assertions that meet the specific characteristics of the program, and then add the assertion detection code to the program. Compared with the code redundancy method, the overhead of assertion detection is significantly reduced. However, the reliability improvement of this method is not obvious, and it is not generally applicable. Some typical examples are the assertion detection method based on variable type proposed by Hiller et al. [13] and the fault screening method proposed by Paul Racunas et al. [28].

Table 1 Example of ThOR redundancy detection methods and Fault Screening assertion detection methods

ThOR redundancy detection method		*Fault Screening assertion detection method*
Original code	*Modified code*	
int a, b;	*int a0, b0, b1, b1;*	*int a, b;*
		a=1;b=1;
a=b;	*a0=b0;*	*assert(a + b==2);*
	a1=b1;	
	if(b0!=b1);	*int c;*
		for(c=1;c<100;c++)
	error();	*c+=a + b;*
a=b + c;	*a0=b0 + c0;*	*assert(c>=0);*
	a1=b1 + c1;	
	if((b0!=b1) || (c0!=c1))	
	error();	

In summary, most previous studies have considered the reliability and performance trade-off only via one detection method, without considering the allocation between different detection methods. Some studies considering the trade-off between different detection methods [36], however, this study focused on the micro-architecture level rather than the software level. One interesting study is a system-level cross-layer early reliability analysis framework called 'SyRA' [38], which considered the application of a cross-layer combination of two protection mechanisms. However, their study focuses on the combination of hardware-based method and software-based method, they still did not consider how to make a trade-off with different software-based detection methods.

3 System Reliability and Overhead Model

In this paper, we present a quantitative evaluation method of the detection method. The reliability and performance trade-off is achieved by assigning code redundancy, assertion detection or without using detection methods to each BBs, as shown in Fig. 1.

In order to achieve a trade-off between system reliability and performance, the first step is to calculate the reliability and overhead of the system. There are many studies in this field. However, since this paper focuses on the trade-off mechanism between reliability and performance, to simplify the calculation process, we adopt a simple reliability analysis model proposed by Savino et al. in [32]. It is worth noting that the accuracy of reliability analysis model will not significantly impact the trade-off mechanism or the final result. We will elaborate on this further in Section 4.

In this paper, we use "R" to represent the reliability of the system, which indicates the probability of the program being

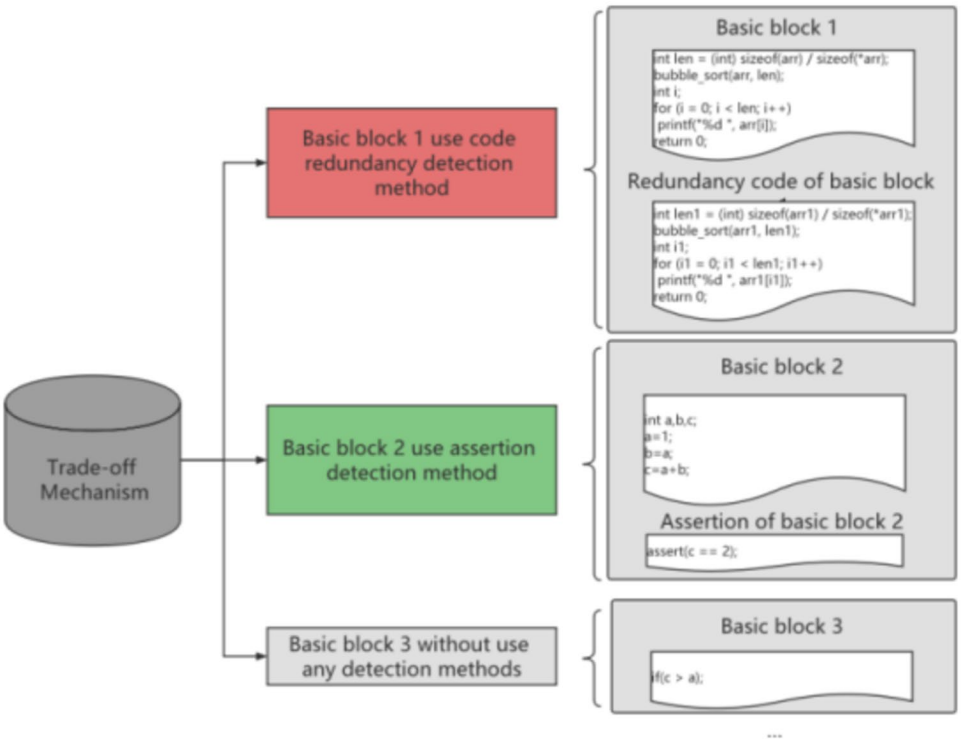

Fig. 1 Example of allocating the apposite method to each BBs

executed correctly. We use "O" to represent the overhead brought by the introduction of detection methods, which is defined as the ratio between the execution time of the program with added detection code and execution time of the original program.

3.1 System Reliability Assessment

We divide a complete program into basic blocks (BBs). BBs are code segments that are executed in sequence with only one parameter entry and exit. In this context, a program comprises n functions denoted as F_i $(1 \le i \le n)$, a function comprises m BBs denoted as BB_{ij} $(1 \le j \le m)$, and a BB comprises I instructions denoted as I_{ijk} $(1 \le k \le l)$. The correct execution of each instruction is an independent event; thus, reliability can be expressed as the probability of several instructions being executed correctly, as follows.

$$R = \prod_{i=1}^{n} P_C(F_i) = \prod_{i=1}^{n}\prod_{j=1}^{m} P_C(BB_{ij}) = \prod_{i=1}^{n}\prod_{j=1}^{m}\prod_{k=1}^{l} P_C(I_{ijk}) \tag{1}$$

where R represents the probability of the program executing correctly, P_C represents the probability of correct execution: $P_C(F_i)$ represents the probability of function F_i executing correctly, $P_C(BB_{ij})$ represents the probability of BB_{ij} executing correctly, and $P_C(I_{ijk})$ represents the probability of instruction I_{ijk} executing correctly.

According to Eq. (1), the reliability of BB is determined by the reliability of each instruction within the BB, and these reliabilities can be obtained through Savino et al.'s statistical systematic reliability assessment study [32].

On this basis, we will further consider the improvement of reliability by detection methods. After considering the detection method, there are two situations that a BB performs correctly: 1) when it executes correctly itself; 2) when it executes incorrectly itself but the detection method detects the error. However, in both cases, it is necessary to ensure that the detection method executes correctly. The reason is if the detection method executes incorrectly, even if the program itself executes correctly, the BB will ultimately execute incorrectly. These are expressed mathematically as follows.

$$P_C(BB_{ij}) = \{P_C(BB_{ij}) + (1 - P_C(BB_{ij})) \times P_{Mdet}\} \times \prod P_C(Id_{ijk}) \tag{2}$$

where P_{Mdet} represents the probability that the detection method can detect an error. Next, we will introduce two reinforcement methods into our reliability calculation formula: we assume that the detection probability using the code redundancy method is 1, and the detection probability using the assertion detection method is P_{Mdet}. We will discuss the P_{Mdet} in detail in Section 4.1.

The set of BBs using the code redundancy method is $A = \{B_{i1,} B_{i2,} \cdots B_{ij,} \cdots\}$ $(1 \le i \le n, 1 \le j \le m)$, according to the hypothesis in the last paragraph, all the reliabilities of BBs that belong to A are 1. The set of BBs using the assertion detection method is $B = \{B_{i1,} B_{i2,} \cdots B_{ij,} \cdots\}$ $(1 \le i \le n, 1 \le j \le m)$. Thus, the reliability after using detection method is expressed as follows.

$$R = \prod_{i=1}^{n} \prod_{j\in A} (1 \times \prod_{j\in B} \{P_C(B_{ij}) + (1 - P_C(B_{ij})) \times P_{Mdet}\} \times \prod_{j\notin A\cup B} P_C(B_{ij}) \times \prod P_C(Id)$$

$$R = \prod_{j\in B} \{P_C(B_{ij}) + (1 - P_C(B_{ij})) \times P_{Mdet}\} \times \prod_{j\notin A\cup B} P_C(B_{ij}) \times \prod P_C(Id) \quad (3)$$

where *Id* represents the additional detection instructions of the detection method, and *PC*(*Id*) represents the probability of *Id* being correctly executed. The symbol '1 ×' is due to we assume that the reliability of the BBs using code redundancy being 100% (1).

3.2 System Overhead Assessment

We define the overhead of system as time overhead, which is the ratio of execution time after using the detection methods to the execution time of source program. We use the clock cycle to represent the execution time.

$$O = T_{det}/T_{source} \quad (4)$$

where O represents the time overhead, T_{det} represents the overhead of the execution time after using the detection methods, and T_{source} represents the execution time of the source program.

The code redundancy method adds redundant code and detection code to the source code, while the assertion method only adds detection code without redundant code. Therefore, the execution time of source code is expressed as follows.

$$T_{source} = \sum_{i=1}^{n} \sum_{j=1}^{m} t_{Bij} \quad (5)$$

where t_{Bij} represents the execution time of a BB without using any detection methods.

The execution time after using detection methods is expressed as follows.

$$T_{det} = \sum_{i=1}^{n} \left\{ \sum_{j\in A} (2 \times t_{Bij} + t_{det}) + \sum_{j\in B} (t_{Bij} + t_{det}) + \sum_{j\notin A\cup B} t_{Bij} \right\} \quad (6)$$

where t_{det} represents the the execution time of the code increased by detection methods.

Thus, from Eqs. (4), (5), and (6), the system overhead O is expressed as follows.

$$O = \left\{ \sum_{i=1}^{n} \sum_{j=1}^{m} t_{Bij} \right\} / \left\{ \sum_{i=1}^{n} \left\{ \sum_{j\in A} (2 \times t_{Bij} + t_{det}) + \sum_{j\in B} (t_{Bij} + t_{det}) + \sum_{j\notin A\cup B} t_{Bij} \right\} \right\} \quad (7)$$

4 Proposed System Reliability and Performance Trade-Off Strategy

In this section, we introduce how to achieving a trade-off between reliability and performance for the entire system. First, we use system reliability analysis tools (such as 'SyRA' [38], 'Flodam' [15], and 'SoftArch' [19]) to calculate the reliability and execution time of the source BB, as well as the reliability and execution time of each BB after using detection methods. Then, we compute the improvement rate of reliability and the increase rate of overhead for each BB. Next, we compute the energy efficiency of different methods and assign the method with higher energy efficiency to each BB. BBs with high reliability do not adopt any detection methods to further reduce overhead Finally, if engineers have different requirements for reliability or performance, we can reassign the result based on these requirements. Figure 2 shows the process flow of our trade-off strategy.

Since the system reliability analysis tool is only used in the first stage (computing reliability and execution time), it is easy to change another tools to increase the accuracy of reliability. Thus, our trade-off strategy is not strongly dependent on the system reliability analysis tool.

4.1 Reliability Improvement Rate

The reliability improvement rate of a BB is defined as follows.

$$R_{imp} = (R_{det}/R_{source}) - 1 \quad (8)$$

where R_{imp} represents the reliability improvement rate of a BB, R_{det} represents the reliability of a BB after using detection methods, R_{source} represents the source reliability.

In combination with Eq. (3), the reliability increases rate R_{imp} for the assertion detection method is given as follows.

$$R_{imp} = \left\{ \left(((P_C(BB_{ij}) + (1 - P_C(BB_{ij})) \times P_{Mdet}) \times P_C(Id)^{\wedge}(et)) / \left(P_C(BB_{ij})^{\wedge}(et)\right) \right) \right\} - 1 \quad (9)$$

where *et* represents the execution time.

For the BBs using the code redundancy method, the reliability increase rate R_{imp} is expressed as follows.

$$R_{imp} = \left\{ P_C(Id)(et)/P_C(BB_{ij})^{\wedge}(et) \right\} - 1 \quad (10)$$

The detection probability of assertion detection (P_{Mdet}) can be calculated according to the concept of error masking parameter [20], which refers to the error detection rate of the detection method. An example of P_{Mdet} calculation is shown here using the detection relationship of equality. The result of correct execution of program is denoted as x, the result of

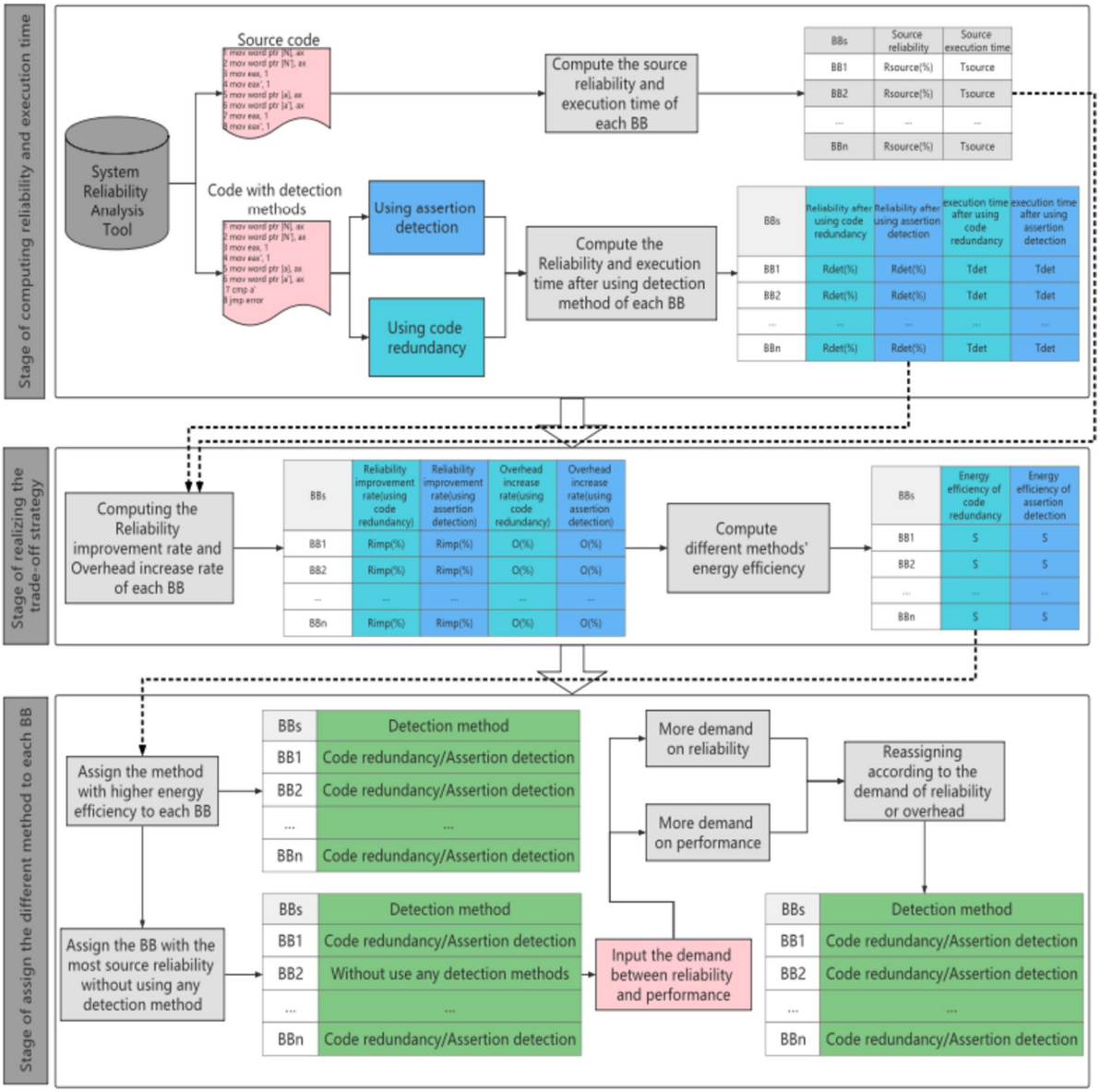

Fig. 2 Trade-off strategy process flow. Note: input (red), output (green), detection methods (blue), computational tasks (light gray)

incorrect execution of program is denoted as x′, and the detection assertion is given as x = a; thus, $x' = x \pm 2k \neq a$. Therefore, x′ = a cannot be satisfied; thus, the P_{Mdet} is 100%. Ref. [36] has provided a detailed approach to computing P_{Mdet}, so we will not provide too many explanations in this paper.

However, most BBs have more than one variable, thus, we must consider how to calculate P_{Mdet} for a BB with multiple variables. In a BB where each variable is executed correctly is an event that is independent of other events, we assume a BB contains n variables. Additionally, the detection probability for each variable is denoted $P_{Mdet}(v_n)$. Then, we can get the P_{Mdet} for the basic block as follows.

$$P_{Mdet} = \prod_{i=1}^{n} P_{Mdet}(v_i) \tag{11}$$

If a BB has two variables: a and b, the P_{Mdet} of a is 0.5, the P_{Mdet} of b is 1, then according to Eq. (11), the P_{Mdet} of this BB will be: P_{Mdet} (a)*P_{Mdet} (b) = 0.5*1 = 0.5.

4.2 Overhead Increase Rate

According to the assumption made in Section 3 (assume that the detection probability P_{Mdet} using the program redundancy method is 100%) and Eq. (7), the overhead increase rate for the BBs using the assertion detection method is expressed as follows.

$$O = \left\{\left(t_{det} + t_{Bij}\right) \times (et)\right\} / \left\{t_{Bij} \times (et)\right\} \quad (12)$$

For BBs using the program redundancy method, the overhead increase rate is given as follows.

$$O = \left\{\left(\mathbf{2} \times t_{det} + t_{Bij}\right) \times (et)\right\} / \left\{t_{Bij} \times (et)\right\} \quad (13)$$

4.3 Energy Efficiency of Detection Method

To configure the detection method, we define the ratio of reliability improvement rate to the overhead increase rate of each BB as the energy efficiency. This means that for a BB, the detection method increases the overhead by a part in exchange for improving the reliability. Higher efficiency means that the detection method incurs a small overhead increase in exchange for a significant improvement in reliability. Note that we can select a more efficient detection method to reduce the overhead of the system on the premise of ensuring high reliability.

According to Eqs. (4) and (8), the energy efficiency of the detection methods is expressed as follows.

$$S = R_{imp}/O = \left\{\left(R_{det}/R_{source}\right) - \mathbf{1}\right\} / \left\{T_{det}/T_{source}\right\} \quad (14)$$

where S represents the energy efficiency of detection method.

4.4 Selection of BBs Without Detection Method

Some BBs will contain fewer instructions and have less execution time, which means they have a high reliability. Therefore, we can choose not to use any detection method for these basic blocks, which will reduce system overhead. We set a BB self-reliability threshold $P_{Br,}$ for BBs with self-reliability greater than P_{Br}, there is no detection method is used Ref. [42] has determined that an effective trade-off between system reliability and performance can be obtained when the P_{Br} value is selected as the top 20% of the basic block's self-reliability. The number of BBs without using the detection method is calculated as follows [42].

$$X = (Number\ of\ all\ basic\ blocks) \times 20\% \quad (15)$$

where X represents the number of BBs without using any detection method.

4.5 Example of Trade-Off Mechanism

To illustrate our trade-off mechanism more clearly, we present an example of using a real program to achieve the trade-off. This program contains 11 BBs and is running on the ARM Cortex-M3 architecture. First, we use a System Reliability Analysis Tool to compute the source reliability (R_{source}), source execution time (T_{source}), as well as the reliability (R_{det}) and execution time (T_{det}) after using detection method. The results are reported in Table 2 and Table 3. Then, we compute the reliability improvement rate (R_{imp}) and overhead increase rate (O) for each method using different detection methods. The results are reported in Table 4. Next, we calculate the energy efficiency (S) of each method and report the results in Table 5. Finally, we assign the detection method with the higher energy efficiency (S) to each BB and select the 20% of BBs with the highest reliability without using any detection methods. The results are reported in Table 6.

4.6 Reassign According to the Demand of Reliability or Overhead

In some cases, engineers may have a higher demand for reliability or performance. They can achieve higher reliability by sacrificing system performance, and vice versa. Ref. [42] realizes this process by simply added or subtracted redundant BBs.

We have designed a reassignment process to make our trade-off mechanism more flexible. We first define two variables that represent the demand of reliability (dor) and demand of performance (dop). The relationship between the two variables is described by Eq. (16), where both variables range from 0 to 2. In the default case, both variables are set to 1, indicating an equal demand for reliability and performance. When one variable increases, the other variable decreases accordingly.

$$dor = \mathbf{2} - dop \quad (16)$$

In order to realize the reassign process, dor is ralated to S_{re}, dop is related to S_{asrt}. We add dor and dop to Eq. (14).

Table 2 The R_{source} and T_{source} of each BB

BBs	R_{source} *(%)*	T_{source} *(Clock cycles)*
BB1	87.69	9
BB2	82.05	21
BB3	78.11	25
BB4	89.07	13
BB5	77.86	26
BB6	69.12	44
BB7	96.69	5
BB8	92.12	6
BB9	91.30	12
BB10	75.53	32
BB11	90.55	12

Table 3 The R_{det} and T_{det} of each BB

BBs	R_{det} *of program redundancy (%)*	R_{det} *of assertion detection (%)*	T_{det} *of program redundancy (Clock cycles)*	T_{det} *of assertion detection (Clock cycles)*
BB1	99.91	98.14	22	13
BB2	99.91	91.24	46	25
BB3	99.91	89.33	54	29
BB4	99.91	97.41	30	17
BB5	99.91	86.37	56	30
BB6	99.82	65.55	184	96
BB7	99.91	99.91	14	9
BB8	99.91	99.12	16	10
BB9	99.91	97.91	28	16
BB10	99.82	88.89	136	72
BB11	99.91	97.83	28	16

$$S_{asrt} = dop \times \left(R_{imp}/O\right) \tag{17}$$

$$S_{re} = dor \times \left(R_{imp}/O\right) \tag{18}$$

where S_{asrt} represents the energy efficiency when using assertion detection method, S_{re} represents the energy efficiency when using redundancy detection method.

We add *dop* to Eq. (15):

$$X' = dop \times (Number\ of\ all\ basic\ blocks) \times \mathbf{20\%} \tag{19}$$

where *X'* represents the number of BBs without using any detection methods after reassignment.

Through analysis of Eqs. (17), (18), and (19), we can infer that when engineers have more demand for reliability, they can input a higher value for *dor*, according to Eq. (16), *dop* will decrease. Since *dor* is related to S_{re} and *X'*, and *dop* is related to S_{asrt}, the value of S_{re} will be higher than the default case. This means that more BBs will be assigned to use the code redundancy method, fewer BBs will be assigned to use the assertion detection method, and less number of BBs will be assigned without using any detection method. Therefore, the system reliability will increase at the expense of performance.

5 Experimental Result

In Section 5.1, we describe how the experiment be set up, Section 5.2 and Section 5.3 compares the reliability and overhead of our method with the previous works, Section 5.4 shows the result of reassignment when the values of *dor* and *dop* are adjusted.

Table 4 The R_{imp} and O of each BB

BBs	R_{imp} *of program redundancy (%)*	R_{imp} *of assertion detection (%)*	O *of program redundancy (%)*	O *of assertion detection (%)*
BB1	13.93	11.92	244.44	144.44
BB2	21.76	11.19	219.05	119.05
BB3	27.91	14.37	216.00	116.00
BB4	12.17	9.36	230.77	130.77
BB5	28.33	10.93	215.38	115.38
BB6	44.41	-5.01	418.18	218.18
BB7	3.33	3.33	280.00	180.00
BB8	8.46	7.59	266.67	166.67
BB9	9.44	7.24	233.33	133.33
BB10	32.16	17.69	425.00	225.00
BB11	10.34	8.05	233.33	133.33

Table 5 The S of each BB

BBs	*S of code redundancy*	*S of assertion detection*
BB1	0.0570	0.0825
BB2	0.0994	0.0940
BB3	0.1292	0.1238
BB4	0.0527	0.0715
BB5	0.1315	0.0948
BB6	0.1062	-0.023
BB7	0.0119	0.0185
BB8	0.0317	0.0456
BB9	0.0404	0.0543
BB10	0.0757	0.0786
BB11	0.0443	0.0603

Table 6 The final assign result of each BB

BBs	*Detection method*
BB1	assertion
BB2	redundancy
BB3	redundancy
BB4	assertion
BB5	redundancy
BB6	redundancy
BB7	without use
BB8	without use
BB9	assertion
BB10	assertion
BB11	assertion

5.1 Experimental Set Up

We chose STM32CubeIDE as the experimental platform, which is an integrated development environment for STM32 products. Through Serial Wire Debug (SWD), we can simulate faults caused by space radiation and observe the current state of the system.

In this experiment, fault injection was performed on a real STM32F103c6t6 device connected to the host PC through SWD. The host PC controlled the process of the STM32F103c6t6 device. To simulate the fault, we used "step into" to iterate through every instruction and changed the value of every register through SWD.

Figure 3 shows the experimental process. The window on the left displays the C code, the window in the middle shows the corresponding assembly code, and the window on the right shows all the registers of the ARM Cortex-M3. We can change the value of these registers through this window. Faults are injected into the program by break-point debugging. By viewing and comparing the final result and the running status of the program, we could judge whether an SDC error occurs, and whether the SDC error is detected or not.

The test program are the program written by ourselves, Matrix Multiplication, Qsort, and Rad2deg from the MiBench test set [12]. Single event setup (SEU) is a common soft error, so our experiment uses SEU as soft error model. We injected 272 SEU errors into each program. According to Ref. [17], the confidence level of our result is 95%, and error margin is 5%.

We selected three detection methods and two trade-off methods to contrast with our trade-off method. The detection methods include EDDDDI, implementing error detection by diverse data and duplicated instructions, ThOR, a source code-level redundancy method, and Fault Screening, which performs value range assertion detection for variables. The trade-off methods include RECCO, which implements trade-off by only detecting the key variables, and partial software protection, which only detects the key code segments.

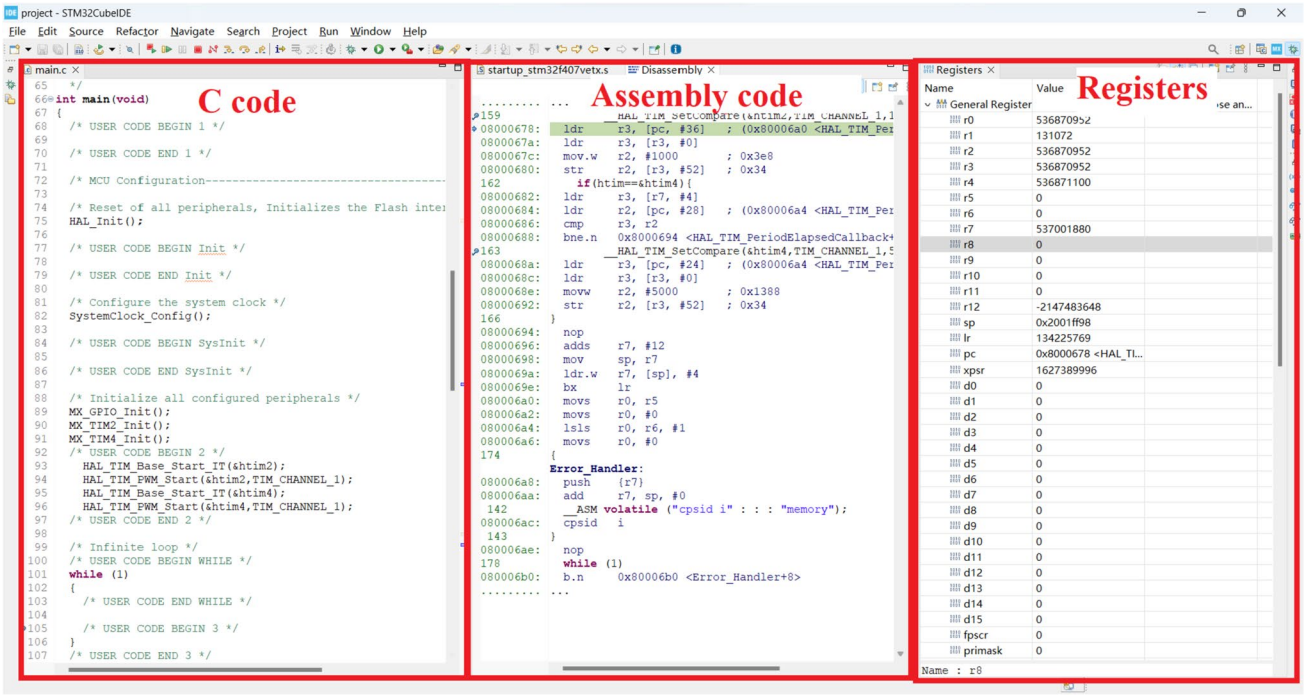

Fig. 3 The experiment platform STM32CubeIDE

5.2 Reliability

We use fault injection to test the reliability of our system. The reliability of some programs has been evaluated in Ref. [34], on this basis, we added our program as a test program. We also added Fault Screening and two trade-off methods, RECCO and partial software protection, to compare with our trade-off method. The result of fault injection is shown in Fig. 4.

Based on the effects of injected faults, we classify the types of errors into the following four categories:

Correct Result (CR): The fault does not change the output.
Hard Fault (HF): The injected fault is detected by default fault exception handlers in the ARM Cortex processor family.
Detected Fault (DF): The injected fault is detected by the detection methods used for data flow error detection.
Undetected Fault (UDF): The injected fault is not detected and it changes the output of the program.

Through analysis of the results, we found that the undetected rates of the two Redundant code methods are both below 2%, the average undetected rate of Fault Screening is 28.15%. The average undetected rate of the two trade-off methods is 9.83%, while the average undetected rate of our method is 14.67%. Our trade-off method uses both Redundant code and assertion detection, and some code segments do not use any detection method, so our undetected rate is 12.67% higher than redundant detection methods and 13.48% lower than assertion detection method. Compared to the two trade-off methods, our trade-off method increased the undetected rate by 4.84%, the reason is in some code segments, we use assertion detection instead of program redundancy.

5.3 Overhead

We use the ratio of clock cycles after using a detection method to the clock cycles of the source program to represent the overhead of the detection method. The result of the overhead is shown in Fig. 5.

Through analysis of the results, we found that due to the coarse-grained of high-level languages, the average overhead of Thor is the highest at 389.5%. The average overhead of EDDDDI is 289.25%. The average overhead of Fault Screening is the lowest at 110.41%. The average overhead of the two trade-off methods is 223.4%, and the average overhead of our method is only 178.06%. Due to the use of two detection methods and some code without any detection method, the overhead of our trade-off method is 45.34% lower than

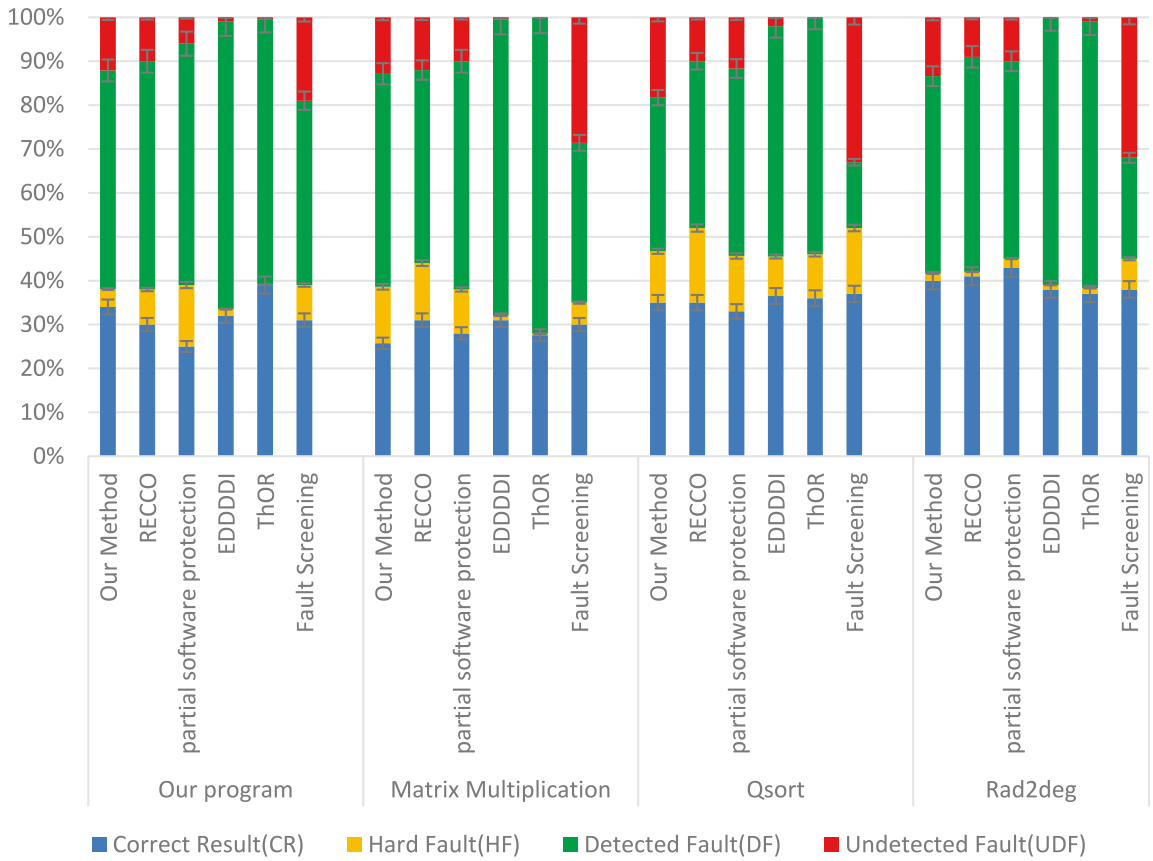

Fig. 4 Result of fault inject result

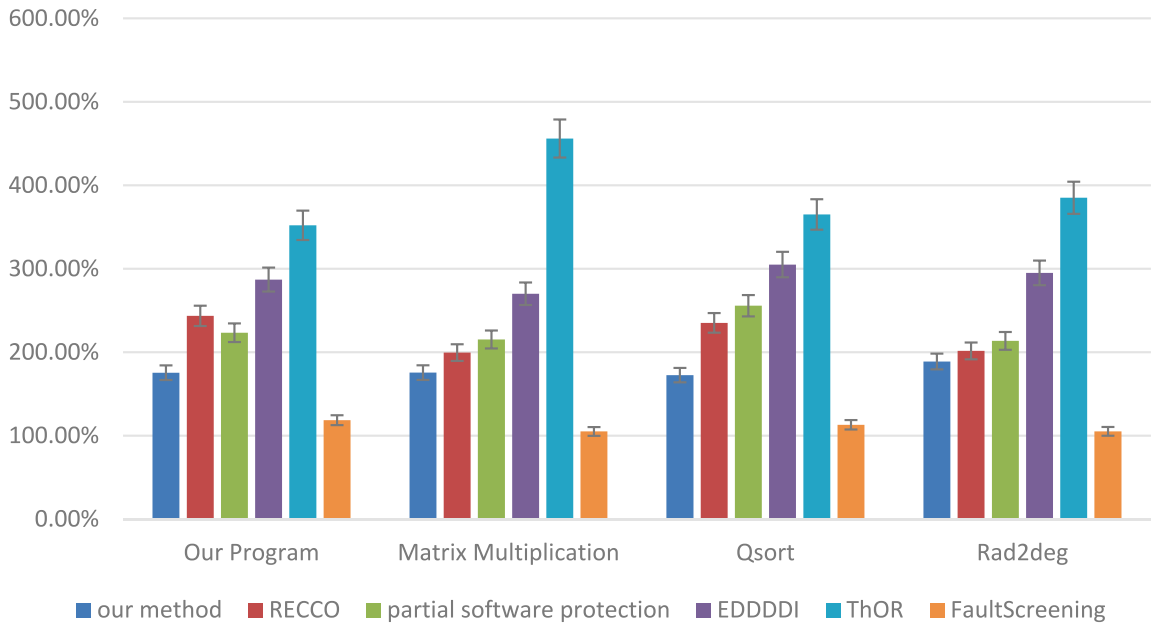

Fig. 5 Result of Overhead

the other two trade-off methods and 61.6% lower than program redundancy methods, which means that our method has a higher performance than other trade-off methods and program redundancy methods.

In order to more clearly compare the advantages of our trade-off method, we use the definition of energy efficiency of detection method in Eq. (14) to characterize the effect of trade-off method. A higher energy efficiency means the trade-off method can bring higher reliability improvement with the same overhead. Therefore, the higher energy efficiency the better effect of trade-off. We calculated the energy efficiency of different methods under different test program. The final results show that the average energy efficiency of our trade-off method is 91.34%, RECCO is 75.42%, partial software protection is 85.82%, EDDDDI is 66.34%, ThOR is 48.49%, and Fault Screening is 73.22%. The energy efficiency of our trade-off method is 15.93% and 5.52% higher than two trade-off methods (RECCO, partial software protection), and 25.01%, 42.85%, 18.12% higher than the detection methods without implement trade-off (EDDDDI, ThOR, Fault Screening). The experimental results are showed in Table 7.

5.4 Results of Reassign

In this section, we demonstrate the final trade-off results when the values of *dop* and *dor* are adjusted. Since fault injection is time-consuming and complex, reliability is measured by the probability of program been executed correctly rather than the detection rate in this section. To illustrate the change in energy efficiency S (see Eq. (14)) when *dor* and *dop* change in our trade-off method. We define the average energy efficiency of each BB as $\overline{S}$:

$$\overline{S} = \frac{\sum S_n}{n} \tag{20}$$

where S_n represents the energy efficiency of the nth BB, n represents the number of BBs.

As shown in Table 8, when *dor* gradually increases and *dop* gradually decreases, reliability and overhead both increase. When *dor* is 0 and *dop* is 2, the reliability of all programs is the lowest, with an average of 27.68%. However, the average overhead is 120.12% which is also the lowest. It is worthy to note that even if *dor* is 0, 60% of BBs will

Table 7 Result of S

Program	Our mesthod	RECCO	Partial software protection	EDDDDI	ThOR	Fault Screening
Our Program	90.38%	82.14%	123.60%	72.95%	44.14%	78.40%
Matrix Multiplication	135.94%	92.17%	102.89%	81.84%	57.60%	78.26%
Qsort	77.35%	66.84%	65.58%	55.24%	48.71%	67.57%
Rad2deg	61.70%	60.52%	51.20%	55.31%	43.52%	68.65%

still be allocated the assertion detection method, so a certain overhead is still needed.

When *dor* and *dop* are both 1, the average reliability increases to 75.66% and the average overhead increases to 178.03%. Compared with *dor* is 0 and dop is 2, the average reliability increases by 47.98%, the average overhead increases by 57.91%. Due to the improvement of *dor*, more BBs use the program redundancy detection methods, because the program redundancy detection methods can bring more reliability improvements.

Finally, when the *dor* is 2 and the *dop* is 0, the average reliability is 95.75%, but the average overhead also increases to 215.72%. This indicates that when *dor* is 2, most of the BBs use the program redundancy detection methods to get the most reliability improvements.

We use $\overline{S}$ to characterize the effect of the trade-off method. When dor increases from 0 to 1, $\overline{S}$ gradually increases, when *dor* and *dop* are both 1, $\overline{S}$ reaches the maximum, when *dor* increases from 1 to 2, $\overline{S}$ gradually decreases. The result shows that when *dor* and *dop* are both 1, our trade-off method achieves the optimal configuration of detection methods for each BBs. The value of $\overline{S}$ is much smaller than the value of S in Section 5.3, because S is the energy efficiency of the overall program, $\overline{S}$ is the average of the energy efficiency of each BB.

In conclusion, when *dor* is 0 and *dop* is 2, the average value of reliability is 27.68%, and the average value of overhead is 120.12%. When *dor* is 2 and *dop* is 0, the average value of reliability and overhead are 95.75% and 215.72%, respectively. $\overline{S}$ represents the average energy efficiency of each BB (see Eq. (20)). The higher $\overline{S}$ is, the better the trade-off effect is. Only when *dor* and *dop* are both 1, $\overline{S}$ reaches the maximum value. The result shows that our trade-off method achieves the optimal configuration of detection methods for each BBs when *dor* and *dop* are both 1.

In the experimental results, we observed a significant difference in program reliability. For example, Qsort program had an initial reliability of 81.989%, while Rad2deg program had an initial reliability of only 0.005%. This is because Rad2deg program has a large number of loop structures that are prone to errors. Conversely, Qsort program is a non-arithmetic program with a high probability of correct execution for each basic block, resulting in higher reliability compared to arithmetic programs.

By reassigning the demand of reliability and performance, engineers have more choices when prioritizing between these two factors. This enables them to better meet the requirements of the actual system.

6 Conclusion

In this paper, we propose a general trade-off method for software-based soft error detection that quantifies the evaluation of detection methods. The trade-off method considers the allocation between code redundancy and assertion detection methods.

We used the ratio of the reliability improvement rate to the overhead increase rate to characterize the effect of trade-off. Experimental results show that the average energy efficiency of our trade-off method is 91.34%, which is 21.49% higher than the other methods. This suggests that our trade-off method works better. Additionally, the trade-off method has flexible scalability and can be executed simultaneously with other methods, such as [16], to further achieve an effective trade-off between the reliability and performance of the system.

Future studies could use the machine learning method [16] to enable a dynamic analysis of the program BB. In this paper, only the data flow detection method was considered, the control flow detection method [35] should be

Table 8 Results of R, O and $\overline{S}$ when the values of *dop* and *dor* are adjusted

dor	*dop*	*Our Program*			*Matrix Multiplication*		
		R	O	$\overline{S}$	R	O	$\overline{S}$
0. 00	2. 00	26.44%	111.38%	5.27%	2.28%	103.62%	6.20%
0.5	1.5	31.24%	114.23%	5.92%	60.12%	170.18%	7.49%
1. 00	1. 00	72.17%	175.59%	14.96%	66.11%	175.17%	7.88%
1.5	0.5	95.65%	215.31%	6.98%	85.87%	209.54%	6.78%
2. 00	0. 00	98.83%	218.50%	7.09%	93.51%	222.00%	6.88%
dor	*dop*	*Qsort*			*Rad2deg*		
		R	O	$\overline{S}$	R	O	$\overline{S}$
0. 00	2. 00	81.98%	138.52%	2.64%	0.00%	126.94%	3.10%
0.5	1.5	90.15%	153.57%	2.76%	31.25%	174.59%	3.36%
1. 00	1. 00	94.15%	172.52%	3.04%	70.20%	188.83%	5.28%
1.5	0.5	96.54%	195.25%	2.54%	88.96%	198.65%	4.15%
2. 00	0. 00	98.83%	217.58%	2.57%	91.82%	204.80%	3.91%

considered in the future. The mathematical model for the reliability evaluation still needs improvement. At present, there have been a large number of related studies, such as the Markov chain model [14, 26], the Petri network model [39, 41], etc. We will also observe the experimental results using different level of fault injection, similar studies can be referred to literature [25, 40].

As the number of human-launched spacecraft increases each year, the chip cost of spacecraft becomes more important to the industry. The previous detection method caused large time overhead, which is unacceptable for embedded space control systems with high real-time performance. Therefore, this paper proposes an effective trade-off strategy to reduce time overhead, which helps to replace high-cost, low-performance aerospace-grade chips with low-cost, high-performance space-class chips in space systems.

Acknowledgements This work was supported in part by National Natural Science Foundation of China (No. 61106029).

Funding The authors declare that they have no known competing financial interests or personal relationships that could have appeared to influence the work reported in this paper.

Data Availability Statement The data that support the findings of this study are available from the corresponding author upon reasonable request.

Declarations

Competing interests The authors declare no competing interests.

References

1. Aranda LA, Reviriego P, Maestro JA (2018) A Comparison of Dual Modular Redundancy and Concurrent Error Detection in Finite Impulse Response Filters Implemented in SRAM-Based FPGAs Through Fault Injection. IEEE Trans Circuits Syst II Express Briefs 65(3):376–380. https://doi.org/10.1109/TCSII.2017.2717490
2. Argyrides C, Ferreira RR, Lisboa CA, Carro L (2011) Decimal Hamming: A Software-Implemented Technique to Cope with Soft Errors. IEEE International Symposium on Defect and Fault Tolerance in VLSI and Nanotechnology Systems 2011:11–17. https://doi.org/10.1109/DFT.2011.35
3. Baumann RC (2005) Radiation-induced soft errors in advanced semiconductor technologies. IEEE Trans Device Mater Reliab 5(3):305–316. https://doi.org/10.1109/TDMR.2005.853449
4. Benso A, Chiusano S, Prinetto P (2000) A software development kit for dependable applications in embedded systems. Proceedings International Test Conference 2000 (IEEE Cat. No.00CH37159) 170–178. https://doi.org/10.1109/TEST.2000.894204
5. Benso A, Chiusano S, Prinetto P, Tagliaferri L (2000) A C/C++ source-to-source compiler for dependable applications. Proceeding International Conference on Dependable Systems and Networks. DSN 71–78. https://doi.org/10.1109/ICDSN.2000.857517
6. Benso A, Di Carlo S, Di Natale G, Prinetto P, Taghaferri L (2003) Data criticality estimation in software applications. International Test Conference, 2003. Proceedings ITC 2003:802–810. https://doi.org/10.1109/TEST.2003.1270912
7. Chukov GV et al (2014) SEE Testing Results for RF and Microwave ICs. IEEE Radiation Effects Data Workshop (REDW) 2014:1–3. https://doi.org/10.1109/REDW.2014.7004589
8. Das A, Touba NA (2019) Efficient One-Step Decodable Limited Magnitude Error Correcting Codes for Multilevel Cell Main Memories. IEEE Trans Nanotechnol 18:575–583. https://doi.org/10.1109/TNANO.2019.2917139
9. Didehban M, Shrivastava A, Lokam SRD (2017) NEMESIS: A software approach for computing in presence of soft errors. IEEE/ACM International Conference on Computer-Aided Design (ICCAD) 2017:297–304. https://doi.org/10.1109/ICCAD.2017.8203792
10. Fleetwood DM (2021) Radiation Effects in a Post-Moore World. IEEE Trans Nucl Sci 68(5):509–545. https://doi.org/10.1109/TNS.2021.3053424
11. González CJ, Chenet CP, Budelon M, Vaz RG, Gonçalez O, Balen TR (2017) Evaluation of a mixed-signal design diversity system under radiation effects. 2017 18th IEEE Latin American Test Symposium (LATS) 1–6. https://doi.org/10.1109/LATW.2017.7906751
12. Guthaus MR, Ringenberg JS, Ernst D, Austin TM, Mudge T, Brown RB (2001) MiBench: A free, commercially representative embedded benchmark suite. Proceedings of the Fourth Annual IEEE International Workshop on Workload Characterization. WWC-4 (Cat. No.01EX538) 3–14. https://doi.org/10.1109/WWC.2001.990739
13. Hiller M (2000) Executable assertions for detecting data errors in embedded control systems. Proceeding International Conference on Dependable Systems and Networks. DSN 24–33. https://doi.org/10.1109/ICDSN.2000.857510
14. Huang Y, Pan X, Hu L (2015) Rapid assessment of system-of-systems(SoS) mission reliability based on Markov chains. First International Conference on Reliability Systems Engineering (ICRSE) 2015:1–6. https://doi.org/10.1109/ICRSE.2015.7366452
15. Kritikakou A, Sentieys O, Hubert G, Helen Y, Coulon JF, Deroux-Dauphin P (2022) Flodam: Cross-Layer Reliability Analysis Flow for Complex Hardware Designs 2022 Design, Automation & Test in Europe Conference & Exhibition (DATE) 819–824. https://doi.org/10.23919/DATE54114.2022.9774541
16. Laguna I, Schulz M, Richards DF, Calhoun J, Olson L (2016) IPAS: Intelligent protection against silent output corruption in scientific applications. IEEE/ACM International Symposium on Code Generation and Optimization (CGO) 2016:227–238
17. Leveugle R, Calvez A, Maistri P, Vanhauwaert P (2009) Statistical fault injection: Quantified error and confidence. 2009 Design, Automation & Test in Europe Conference & Exhibition, pp. 502-506. https://doi.org/10.1109/DATE.2009.5090716
18. Li J, Reviriego P, Xiao L, Wu H (2021) Protecting Memories against Soft Errors: The Case for Customizable Error Correction Codes. In IEEE Trans Emerg Topics Comput 9(2):651–663. https://doi.org/10.1109/TETC.2019.2953139
19. Li X, Adve SV, Bose P, Rivers JA (2005) SoftArch: an architecture-level tool for modeling and analyzing soft errors. 2005 International Conference on Dependable Systems and Networks (DSN'05) 496–505. https://doi.org/10.1109/DSN.2005.88
20. Ma J, Duan Z, Tang L (2019) A Methodology to Assess Output Vulnerability Factors for Detecting Silent Data Corruption. IEEE Access 7:118135–118145. https://doi.org/10.1109/ACCESS.2019.2936893
21. Mittal S, Vetter JS (2016) A Survey of Techniques for Modeling and Improving Reliability of Computing Systems. In IEEE Trans Parallel Distrib Syst 27(4):1226–1238. https://doi.org/10.1109/TPDS.2015.2426179

22. Nahmsuk O (2001) Software implemented hardware fault tolerance. Ph.D. Dissertation. Stanford University, Stanford, CA, USA. Advisor(s) Edward J Mccluskey
23. Oh N, Shirvani PP, McCluskey EJ (2002) Error detection by duplicated instructions in super-scalar processors. IEEE Trans Reliab 51(1):63–75. https://doi.org/10.1109/24.994913
24. Oh N, Mitra S, McCluskey EJ (2002) ED/sup 4/I: error detection by diverse data and duplicated instructions. IEEE Trans Comput 51(2):180–199. https://doi.org/10.1109/12.980007
25. Palazzi L, Li G, Fang B, Pattabiraman K (2019) A Tale of Two Injectors: End-to-End Comparison of IR-Level and Assembly-Level Fault Injection 2019 IEEE 30th International Symposium on Software Reliability Engineering (ISSRE) 151–162. https://doi.org/10.1109/ISSRE.2019.00024
26. Pham T, Defago X (2013) Reliability Prediction for Component-Based Software Systems with Architectural-Level Fault Tolerance Mechanisms 2013 International Conference on Availability, Reliability and Security 11–20. https://doi.org/10.1109/ARES.2013.8
27. Philip Shirvani P, Edward McCluskey J (1998) Fault-Tolerant Systems in A Space Environment: The CRC ARGOS Project. CRC Tech Rep No. 98-2 (CSL TR No. 98-774)
28. Racunas P, Constantinides K, Manne S, Mukherjee SS (2007) Perturbation-based Fault Screening 2007 IEEE 13th International Symposium on High Performance Computer Architecture 169–180 https://doi.org/10.1109/HPCA.2007.346195
29. Raji M, Sabet MA, Ghavami B (2019) Soft Error Reliability Improvement of Digital Circuits by Exploiting a Fast Gate Sizing Scheme. IEEE Access 7:66485–66495. https://doi.org/10.1109/ACCESS.2019.2902505
30. Rebaudengo M, Reorda MS, Violante M, Torchiano M (2001) A source-to-source compiler for generating dependable software. Proceedings First IEEE International Workshop on Source Code Analysis and Manipulation 33–42. https://doi.org/10.1109/SCAM.2001.972664
31. Reis GA, Chang J, Vachharajani N, Rangan R, August DI (2005) SWIFT: software implemented fault tolerance. Int Symp Code Gen Opt 243–254. https://doi.org/10.1109/CGO.2005.34
32. Savino A, Di Carlo S, Politano G, Benso A, Bosio A, Di Natale G (2012) Statistical Reliability Estimation of Microprocessor-Based Systems. IEEE Trans Comput 61(11):1521–1534. https://doi.org/10.1109/TC.2011.188
33. Shoji T, Nishida S, Hamada K, Tadano H (2015) Cosmic ray neutron-induced single-event burnout in power devices. IET Power Electron 8:2315–2321. https://doi.org/10.1049/iet-pel.2014.0977
34. Thati VB, Vankeirsbilck J, Penneman N, Pissoort D, Boydens J (2018) 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 2018). Association for Computing Machinery, New York, NY, USA, Article 23, 1–9. https://doi.org/10.1145/3230833.3230854
35. Thati VB, Vankeirsbilck J, Pissoort D, Boydens J (2019) Hybrid Technique for Soft Error Detection in Dependable Embedded Software: a First Experiment. IEEE XXVIII International Scientific Conference Electronics (ET) 2019:1–4. https://doi.org/10.1109/ET.2019.8878497
36. Tselonis S, Kaliorakis M, Foutris N, Papadimitriou G, Gizopoulos D (2016) Microprocessor reliability-performance tradeoffs assessment at the microarchitecture level 2016 IEEE 34th VLSI Test Symposium (VTS) 1–6. https://doi.org/10.1109/VTS.2016.7477300
37. Turner JB, Agardy FJ (1994) The Advanced Research and Global Observation Satellite Program (ARGOS). Proc Space Prog TechnolConf 1994–4580
38. Vallero A et al. (2019) SyRA: Early System Reliability Analysis for Cross-Layer Soft Errors Resilience in Memory Arrays of Microprocessor Systems. In IEEE Transactions on Computer 68(5):765–783. https://doi.org/10.1109/TC.2018.2887225
39. Wang Y, Li M, Li L (2013) The Research of System Reliability Calculation Method Based on the Improved Petri Net. Int Conf Inf Technol Appl 2013:279–281. https://doi.org/10.1109/ITA.2013.72
40. Wei J, Thomas A, Li G, Pattabiraman K (2014) Quantifying the Accuracy of High-Level Fault Injection Techniques for Hardware Faults 2014 44th Annual IEEE/IFIP International Conference on Dependable Systems and Networks 375–382. https://doi.org/10.1109/DSN.2014.2
41. Wu X, Yu H (2019) A Petri Net Modeling Approach for Reliability of PMS with Time Redundancy 2019 IEEE 19th International Conference on Software Quality, Reliability and Security Companion (QRS-C) 249–254. https://doi.org/10.1109/QRS-C.2019.00055
42. Xiong L, Tan Q (2011) A Configurable Approach to Tolerate Soft Errors via Partial Software Protection. IEEE Ninth International Symposium on Parallel and Distributed Processing with Applications Workshops 2011:260–265. https://doi.org/10.1109/ISPAW.2011.45

Zhenyu Zhao is currently working toward the B.S. degree in Microelectronics Science and Engineering with the Department of Electronic and Information Engineering, Nanjing University of Aeronautics and Astronautics, Nanjing, China.

Xin Chen received the B.S. degree in Electronic Science and Technology from Southeast University, Nanjing, China, in 2005, the PhD degree in Microelectronics and solid-state electronics from Southeast University, Nanjing, China, in 2010.

Yufan Lu received the B.S. degree in Electronic Information Engineering from Nanjing Tech University, Nanjing, China, in 2016, the M.S. degree in Circuit and System from Nanjing University of Aeronautics and Astronautics, Nanjing, China, in 2019, and the Ph.D. degree in Computing and Electronic Systems from the University of Essex, United Kingdom, in 2023.