ASTES

Advances in Science, Technology and Engineering Systems Journal Vol. 2, No. 3, 623-633 (2017)

www.astesj.com

Special Issue on Recent Advances in Engineering Systems

ASTESJ

ISSN: 2415-6698

Data Error Detection and Recovery in Embedded Systems: a Literature Review

Venu Babu Thati*[1], Jens Vankeirsbilck[1], Jeroen Boydens[1], Davy Pissoort[2]

[1]*Reliability in Mechatronics and ICT, Department of Computer Science, KU Leuven, 8200 Brugge, Belgium*

[2]*Reliability in Mechatronics and ICT, Department of Electrical Engineering, KU Leuven, 8200 Brugge, Belgium*

ARTICLE INFO

Article history:
Received :05 April, 2017
Accepted :11 May, 2017
Online: 04 June, 2017

Keywords:
Data flow errors
Error detection
Error recovery
Reliability
Embedded systems

ABSTRACT

This paper presents a literature review on data flow error detection and recovery techniques in embedded systems. In recent years, embedded systems are being used more and more in an enormous number of applications from small mobile device to big medical devices. At the same time, it is becoming important for embedded developers to make embedded systems fault-tolerant. To make embedded systems fault-tolerant, error detection and recovery mechanisms are effective techniques to take into consideration. Fault tolerance can be achieved by using both hardware and software techniques. This literature review focuses on software-based techniques since hardware-based techniques need extra hardware and are an extra investment in cost per product. Whereas, software-based techniques needed no or limited hardware. A review on various existing data flow error detection and error recovery techniques is given along with their strengths and weaknesses. Such an information is useful to identify the better techniques among the others.

1. Introduction

In general, a critical aspect of any computer system is its reliability. Computers are expected to perform tasks not only quickly, but also correctly [1]. Recent trends in embedded systems attract industries to use them more and more in day-to-day life for an increasing number of applications. Application areas include, but are not limited to, mechatronic industries, medical equipment, smart energy consumers, mobility. Reduced size and reduced supply voltage make systems more susceptible to disturbances. Since there are more systems in use and the environment becomes more harsh, a system failure or a system crash is more likely to occur. A system failure could lead to serious consequences such as human injury, environment pollution and a huge amount of money loss for industries [2].

The rise in usage of electronics under harsh conditions significantly increases the probability of all kinds of disturbances from the environment. Such disturbances are glitches, electromagnetic interference, temperature variations, etc. [3–6]. It is proven that decreasing the size and supply voltage of the components in circuits and increasing their complexity leads to less reliable systems [7]. The corresponding systems are susceptible to soft errors (bit flips) which are typically transient. Transient faults do not cause any permanent physical damage and can be restored by overwriting the introduced bit flip or by a system restart. Still these faults are categorized as systematic faults since given the exact same circumstances these faults will re-appear in exactly the same way. Because the environment changes, transient faults don't occur continuously, unlike design and manufacturing faults [1,8,9].

Errors in embedded systems can cause unusual behavior and degrade systems integrity and reliability [7]. A number of hardware and software techniques have been developed to make embedded systems fault-tolerant against transient faults [10,11]. Fault tolerance is a two step process. The first step is fault detection, indicating that somewhere in the system fault has occurred. The second step in the process is fault recovery, restoring the system from fault state to the normal state [12].

Today, fault tolerance is mainly achieved via hardware solutions. Such hardware-based solutions are hardware redundancy approaches to meet the requirements of the reliability. Such hardware redundancy techniques are expensive since they have to be implemented on every product produced. A commonly used hardware-based technique for error detection in embedded systems is N-modular redundancy. This technique uses N (N>2) parallel modules for comparing the original and redundant process results. This hardware redundancy technique introduces a 100* (N- 1)% performance and memory overhead but does achieve a fault coverage of 100% [12,13]. To reduce the overheads in hardware-based fault tolerance techniques different software-based redundant techniques have been proposed and implemented [12,13]. Such software solutions would lead to a more cost-efficient solution in many situations. Due to its

*Venu Babu Thati, KU Leuven; Spoorwegstraat 25; 8200 Brugge, Belgium
Contact No: +32 (50)664805 & venubabu.thati@kuleuven.be

https://dx.doi.org/10.25046/aj020380

flexibility and cost, software-based solutions are used in a number of applications. Software redundancy increases the system's reliability but requires extra memory space and processing time to execute redundant instructions [15]. A number of software-based data error detection and recovery techniques have been proposed and implemented in literature for fault-tolerant embedded systems [1,7,8,12–22].

According to recent studies, soft errors are one of the primary sources of failure in embedded systems [7,16,23–25]. These soft errors (bit flips) may further affect the system during program execution, leading to a faulty system. Such bit flips have an effect on data flow or control flow of the program. Generally, data flow errors lead to corruption of variables in the program causing a wrong intermediate or output result. In contrast, control flow errors lead to a jump in the program execution order [26–29]. This review paper focuses on various data flow error detection and recovery techniques existing in literature to make embedded systems fault-tolerant against bit flips. Since a number of data error detection and recovery techniques exist in literature, it is important to review and identify the strengths and weaknesses of each of these techniques for a fault-tolerant embedded system. Figure 1 gives an overview of the software-based data protection techniques that will be discussed in this paper.

The remainder of this paper is organized as follows: Section 2 describes and reviews the different data flow error detection techniques. Section 3 describes and reviews the different error recovery techniques. Section 4 provides future work plans and Section 5 concludes this paper.

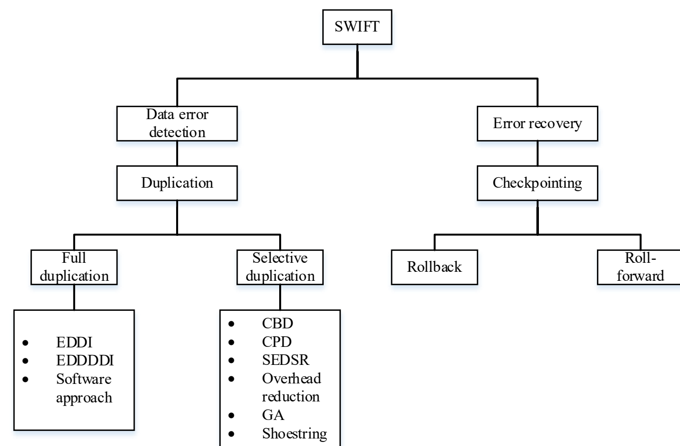


Figure 1. Overview of software-based data protection techniques.

2. Related Work

Soft errors usually occur due to heavy radiation, power supply distortion, environmental temperature fluctuations, and other factors. The introduced soft errors can corrupt the data of the program in execution. To counter this data corruption, a number of data flow error detection and recovery techniques have been proposed in literature [1,7,8,12–17,19–21]. In previous work [4], we listed several data flow error detection techniques and discussed their reported results. The contribution of this review paper is 1) that we list not only detection techniques but also recovery techniques, 2) that we discuss the considered techniques more in depth and 3) that we give several strengths and weaknesses per technique. The provided strengths and weaknesses have been determined based on the technique's inner working and reported results. By looking at the strengths and weaknesses of each of the technique presented under error detection and error recovery, one can easily identify the better technique immediately with clear reasoning.

3. Data Flow Error Detection

This section presents and reviews various existing data flow error detection techniques such as EDDI (Error Detection by Duplicated Instructions), ED^4I (Error Detection by Diverse Data and Duplicated Instructions), Software approach, CBD (Critical Block Duplication), CPD (Critical Path Duplication), SEDSR (Soft Error Detection using Software Redundancy), Checking rules, GA (Genetic Algorithm) and Shoestring approach. Strengths and weaknesses of each of these techniques will be discussed. All of the presented data flow error detection techniques are software-based.

Duplication is the basic mechanism involved in data error detection techniques [4]. A number of data flow error detection techniques have been developed based on a unique duplication mechanism for better fault coverage or lower overhead in terms of memory consumption. The duplication can be applied at various levels such as a full duplication and selective duplication [1,4,7,8,12–15,17,18]. Full duplication techniques and selective code duplication techniques are discussed in Sections 2.1 and 2.2 respectively.

In order to evaluate the data error detection techniques, authors of the corresponding techniques [1,7,12–17] have chosen various case studies for the experiments. Bubble sort, quicksort, insertion sort, and matrix multiplication are the most used case studies in previous research in this field [13,16,17,26,30]. Of course, some of the techniques uses other case studies such as FFT, differential equation solver, mean, vortex, etc. Further, a fault injection mechanism has been used to inject the faults in hardened case studies for validation. All of the provided information in Table 1 and Table 3 such as error detection techniques, case studies, injected faults, detected faults, fault coverage, performance overhead, and memory overhead are considered from literature [1,7,12 – 17]. Fault coverage, performance overhead, and memory overhead are defined in (1), (2), and (3).

$$Fault\ coverage = \frac{Number\ of\ detected\ faults}{Number\ of\ injected\ faults} \quad (1)$$

$$Memory\ overhead = \frac{Final\ memory - Initial\ memory}{Initial\ memory} \quad (2)$$

$$Performance\ overhead = \frac{Final\ time\ - Initial\ time}{Initial\ time} \quad (3)$$

3.1. Full Duplication

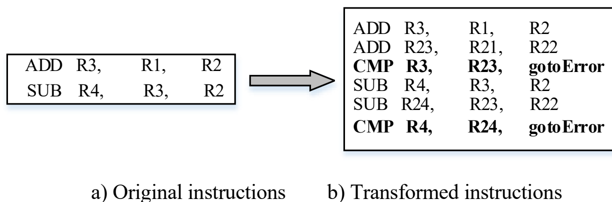
This section presents various existing full duplication techniques for data flow error detection. The basic mechanism involved in all of the full duplication techniques is duplicating the entire code and comparing the original and duplicated output to detect errors. Full code duplication has been performed in different ways for different techniques as in [7,13,15,31].

Table 1. Representation of error detection technique with case studies, case studies length, injected faults and detected faults from literature [1,7,12–17].

Error detection technique	Case studies used	Case studies length (in lines)	Injected faults	Detected faults
EDDI	Insertion sort Quicksort Matrix mul.	30 36 47	500 500 500	495 491 496
ED^4I	Quick sort Matrix mul. Insertion sort	47 36 30	500 500 500	479 481 482
Software approach	Constant modulus algorithm	20	19520	19520
CBD	Bubble sort Matrix mul. Quick sort	25 36 47	784 784 784	615 561 553
CPD	Differential equation solver	30	784	575
SEDSR	Bubble sort Matrix mul. Quick sort	25 36 47	1000 1000 1000	950 940 956
Checking rules	Bubble sort Matrix mul. Dijkstra's	25 36 19	10000 10000 10000	9560 9600 9350
GA	Bubble sort Regression Quicksort	25 23 47	40000 40000 40000	32800 32000 33600
Shoestring	Vortex Crafty Gap	SPEC2000 benchmark suit	12000 12000 12000	11050 6000 9000

Error detection by duplicated instructions

EDDI is one of the most often used error detection techniques in research [12]. The EDDI technique states three different instructions for program execution: a master instruction (MI), a shadow instruction (SI), and a comparison instruction (CI) [13], as shown in Figure 2b. Figure 2a represents two different master instructions. The master instruction is the original instruction of the source code, while the shadow instruction is the duplicated instruction added to the source code. Validation of correct operation is accomplished by comparing registers and memory values of master instructions and those of shadow instructions. In Figure 2b, first three instructions refer to MI, SI, and CI. If there is any mismatch between the master and shadow output, the comparison instruction reports an error. To achieve the highest fault detection ratio, EDDI is applied at the assembly level [8,13]. In order to evaluate the effectiveness of the proposed technique, quicksort, matrix multiplication, insertion sort, and FFT were used as case studies.



a) Original instructions b) Transformed instructions

Figure 2. Master, shadow, and comparison instructions representation.

Strengths

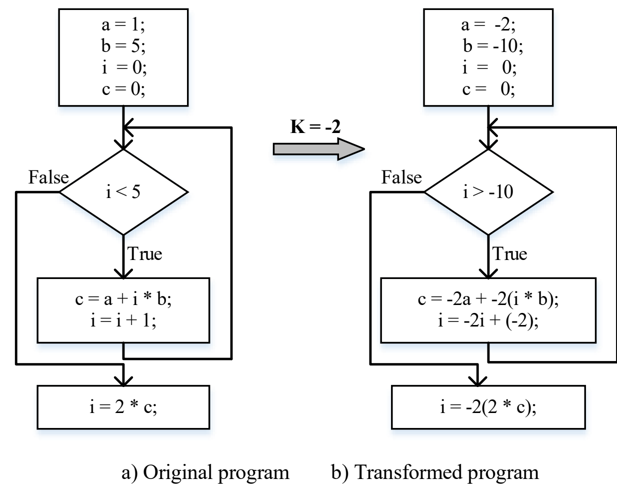
With the final computation results from MI and SI in the program, the error can be detected by placing a comparison instruction. The EDDI technique achieves nearly 98.8% [13,16] of fault coverage by placing a redundant comparison instruction after each MI and SI in the program. The remaining percentage of undetected errors comes from the faults that create an infinite loop in the program. EDDI is one of the techniques that has highest fault coverage in this field.

Weaknesses

Since it is a full duplication technique, all of the instructions presented in the program need to be duplicated. Next to the original (MI) and duplicated (SI) instructions, a comparison instruction has to be placed to report errors. Since each of the original instructions is converted to three instructions, so performance and memory overhead of EDDI are 104.7% and 200% [13,16], as shown in Table 3.

Error detection by diverse data and duplicated instructions

ED^4I detects errors by executing two different programs, the original and a transformed (duplicated) program, and comparing their results. The comparison gives an error if the original and duplicated programs do not lead to the same result. The transformation of ED^4I technique representation is shown in Figure 3, where $x' = k.x$ for integer numbers is used. Where k is the fault detection probability of the program, $\boldsymbol{x}$ is the original program and $\boldsymbol{x'}$ is the transformed program [7]. In the presented ED^4I technique, the optimum value for k that maximizes the fault coverage probability is calculated. After performing the validations with case studies, the authors identified that $k = -2$ is the optimum value to have a maximum fault detection. EDDI and ED^4I techniques are comparable because of their common case stuides.



a) Original program b) Transformed program

Figure 3. Original and transformed program with k = -2.

Strengths

The ED^4I technique presents a transformation algorithm for the program that transforms an original program (integers or floating point numbers) x to a new program $\boldsymbol{x'}$ with diverse data. This technique achieves a 96.1% [7,16] fault coverage when using the optimum value for k. This result is approximately equal to that of the EDDI technique, as shown in Table 3.

Weaknesses

As in EDDI, ED^4I also needs to duplicate the entire original program with diverse data. This technique requires a number of redundant instructions for duplication and comparison, which causes an increment in overhead. The performance and memory overhead imposed by this technique is nearly 126.6%

and 160% [16], as given in Table 3. ED^{4}I transformation is only good for either integers or floating point numbers, but not for both. For example, if a program has mixed data types such as floating point numbers and integers, in that case, we need multiple transformations with different *k* values for each type. The drawback of such a multiple transformations is that it will introduce a more performance and memory overhead.

Detecting soft errors by a pure software approach

The error detection mechanism in the proposed technique is based on a set of transformation rules. These transformation rules are classified into three basic groups: 1) error affecting data, 2) error affecting basic instructions, and 3) error affecting control instructions to detect the errors [31]. Error affecting data rules are used to detect data flow errors, whereas error affecting basic and control instructions rules are used for control flow error detection. In error affecting data, the motive is to identify and define a dependency relationship between variables of the program. Furtherly, classifying them into intermediary variables and final variables based on their role in the program [31]. From Figure 4a, variables *x*, *y*, and *z* are classified as intermediary variables, which are used for other variables operation. Whereas variable *P* is classified as a final variable, which does not participate in any operations. After each write operation to the final variables, both original and duplicated values are compared for a consistency check, if any inconsistency is identified an error detection procedure is activated. By applying the different transformation rules to each of the original variables, this technique is able to detect errors that occur in data, basic instructions, and control instructions.

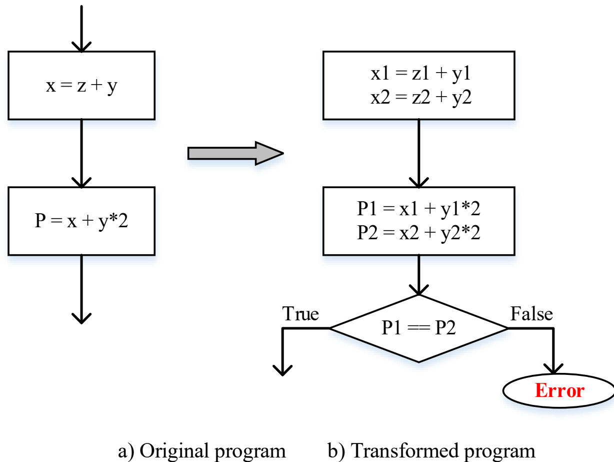


Figure 4. Transformations rules targeting error affecting data in a sample program [31].

Strengths

The presented technique is mainly based on a set of transformation rules. Error affecting data rules are used to detect data flow errors with full duplication scheme. Fault coverage achieved with this technique is 100% [31], because of duplicating the entire program and comparison after each write operation to the final variables. Software approach is one of the techniques that has highest fault coverage in this field.

Weaknesses

Usage of more redundant instruction for duplication and the comparison lead to increase in overhead. The appeared performance and memory overhead in this technique are 244% and 362% [31], as shown in Table 3.

3.2. Selective Duplication

This section presents a number of existing selective code duplication techniques for data flow error detection. The main difference with full duplication techniques is that selective duplication techniques first analyze the program to detect the most important parts and only duplicate those important parts. Defining and identifying the important part of a program can be done in different ways, leading to different techniques [1,12,14–17].

Error detection by critical block duplication

The presented selective code duplication technique is named CBD. The CBD technique follows three different steps to detect data flow errors. The first step is, to identify the critical blocks in the control flow graph. These critical blocks are the most vulnerable in the program because its output has an influence on the other blocks. The second step is to duplicate the identified critical blocks. The final step is to compare the original and duplicated critical blocks to detect errors. The authors of this proposed technique introduced a simple way for critical block detection from the example of control flow graph, as shown in Figure 5. A block that has the most number of outgoing edges to the other blocks in the control flow graph is considered, as a critical block [12]. In Figure 5, block 1, has three outgoing edges to the other blocks, whereas others have less than three. In this case, the highlighted block 1 is identified as a critical block. Furthermore, the critical block, block 1, is duplicated and compared to the original block. If any mismatch is identified between the original and duplicated instructions, it will report an error.

Strengths

In Section 2.1, we have reviewed various full duplication techniques and their advances in fault coverage. But, because of increased performance and memory overhead, it seems that full code duplication is not a good option. Limiting the code duplication scope is useful in real-time and general purpose applications where cost is the primary factor. In CBD, performance and memory overhead are decreased because they use less redundant instructions for duplication and comparison. The appeared performance and memory overhead in this technique are 50% and 101.6% [12], as shown in Table 3.

Weaknesses

With regards to CBD, redundant instructions are inserted only in the critical block, so there is a possibility of missing undetected errors in other blocks leads to a reduction in fault coverage. The achieved fault coverage with CBD technique is only 73.5% [12]. Another major drawback of this technique is that it is a compiler and/or case study dependent and could just act as a full duplication technique. For example, let's consider a control flow graph with 8 basic blocks, 5 of them have two outgoing branches and other 3 have only one. Since 5 of them are critical blocks, they are expected to have at least 80% of the code. Such a control flow graph with CBD approach is duplicating 80% of the original program.

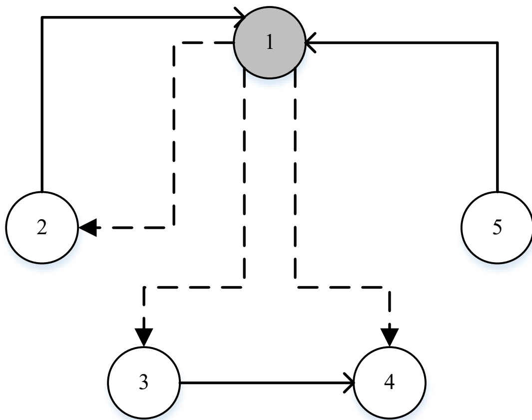


Figure 5. Example of control flow graph with CBD [12].

Error detection by critical path duplication

In this CPD technique, the data flow graph is used instead of using the control flow graph. The data flow graph is used to derive the interconnection of variables and their dependencies and effects on each other. In a data flow graph example, nodes represent the operands and vertices represent the variables of the program, as shown in Figure 6 [14]. The basic idea behind this technique is to identify and duplicate the critical path in the data flow graph. The first step is, to identify the critical path in the data flow graph. The authors of [14,17,32] who proposed CPD technique introduced a simple principle for critical path detection. The longest path in the data flow graph is considered as a critical path because of the great possibility of error occurrence on that long path. According to the principle proposed by the authors, the longest path in the data flow graph is identified and kept in the box, as shown in Figure 6. Next, the identified critical path will be duplicated and the comparison instructions will be placed after each write operation in the final variables. If the final variables in the program are not equal to each other, it reports an error.

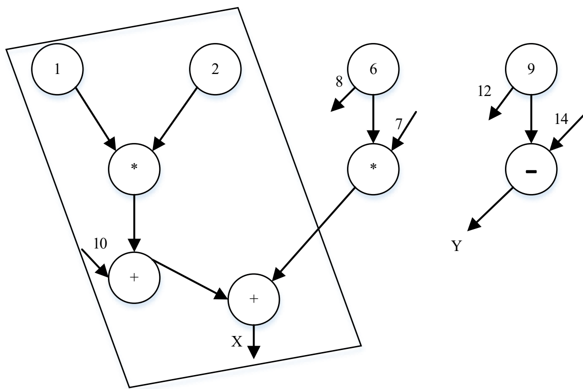


Figure 6. Example of data flow graph with CPD [14].

Strengths

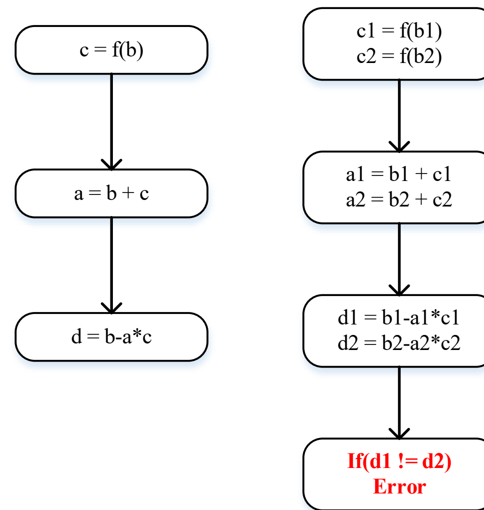
With regard to CPD, only instructions presented in the critical path are duplicated to detect the data flow errors with minimum overhead. In CPD, performance and memory overhead are decreased because they use less redundant instructions for duplication and comparison. The appeared performance and memory overhead in CPD are 60% and 103% [14], as shown in Table 3.

Weaknesses

As far as CPD is concerned, redundant instructions are inserted only in the critical path, so there is a possibility of missing out on the undetected errors in the other small paths in the data flow graph. This lead to reduce in a fault coverage. Fault coverage achieved with CPD is only 73.3% [14]. In CPD, creating a data flow graph is much harder in assembly than C and C++. It is also difficult to perform the duplication and maintain the control flow graph in order when only data flow graph is given.

Soft error detection using software redundancy

This technique is named SEDSR. In this technique, the critical block is duplicated as in CBD. As in [12,17], the critical block is the block with the most number of outgoing edges to the other blocks in the control flow graph, as shown in Figure 5. In this technique, critical block variables are further divided into two categories: (1) middle variables: important in computing the other variables and (2) final variables: they don't perform any computations [17]. In the critical block, a redundant comparison instruction is placed after the final variables to compare these parameters in original and duplicated blocks. Figure 7a represents the sample (original) program of the critical block and variables *a, b,* and *c* are considered as the middle variables and *d* is considered as the final variable. Figure 7b is the duplicated version of a sample program with comparison instruction for the critical block. If any mismatch between original and duplicated variables is identified during the comparison, an error is reported and the program execution is halted. SEDSR and CBD techniques are comparable because of their common case studies.



a) Sample program b) Duplicated program

Figure 7. Sample program of critical block with duplication and comparison [17].

Strengths

SEDSR is one of the critical block duplication techniques. In comparison with CBD, in this technique, the critical block variables are not directly duplicated but furtherly classified into two types such as middle variables and final variables. In SEDSR, by placing a comparison instruction after writing to the classified final variables detect a lot of errors. Fault coverage achieved with this technique is 94.85% [17], which is increased in comparison to the CBD, as shown in Table 3.

Weaknesses

As in CBD, this technique also considers only the critical block with further improvements in the process over CBD, as mentioned in strengths. At the same time, performance and memory overhead are increased because of placing extra comparison instructions. The imposed performance and memory overhead in SEDSR are 112.3% and 134.6% [17], they are increased in comparison to the CBD. Since SEDSR has a similar kind of duplication mechanism as in CBD, this technique is also a compiler and/or case study dependent and could just act as a full duplication.

Overhead reduction in data- flow software-based fault tolerance techniques

The aim of this technique is to provide low overhead with the same level of reliability as in EDDI, ED^4I, and Software approach [7,13,31]. This technique provides an alternative implementation of software-based techniques. The alternative overcomes the drawback of the massive overhead introduced by other techniques for soft error detection. In the presented technique, a set of rules for the data protection are explained as shown in Table 2 such as, 1) global rules: each register used in the program should have its replica, 2) duplication rules: (a) duplicating all instructions except branches, (b) duplicating all instructions except branches and stores, and 3) checking rules: to compare the values of a register with its replica at different positions [15]. Such rules are applied to various methods with the choice to detect the errors. Table 2 explains the purpose of each rule. Checking rules (overhead reduction), EDDI, and ED^4I techniques are comparable because of their common case studies.

Table 2. Checking rules description [15].

Global rules	(Valid for all techniques)
G1	Each register used in the program has a spare register as a replica
Duplicated rules	Performing the same operation on the registers replica
D1	All instructions except branches
D2	All instructions except branches and stores
Checking rules	Compare the value of a register with its replica
C1	Before each read on the register
C2	After each write on the register
C3	Before loads, the register that contains the address
C4	Before stores, the register that contains the data
C5	Before stores, the register that contains the address
C6	Before branches

Strengths

Different methods are implemented in this technique by considering the choice of duplication and checking rules, as shown in Table 2. A couple of methods have equal fault coverage with changes in overhead. Selecting the right checking rules is important because they have an influence on fault coverage and overhead. The method with highest fault coverage and lower overheads is considered as the best method. In this technique, the best method has a fault coverage of 95% with performance and memory overhead of 72.3% and 82% [15]. The appeared performance and memory overhead in this technique are decreased in comparison to the EDDI, and ED^4I techniques, as shown in Table 3.

Weaknesses

Compared to full duplication error detection techniques such as EDDI, and ED^4I, this technique has slightly reduced fault coverage. Fault coverage achieved with this technique is 95% because of using less number of redundant instructions with the choice of checking rules in comparison to the EDDI and ED^4I techniques.

Method for hardening a program against soft error using genetic algorithm

In this technique, GA has been used to identify the most vulnerable blocks of the program through input data [16]. The identified vulnerable blocks have to be strengthened against errors through duplication and comparison. The proposed technique follows three different steps to detect errors as shown in Figure 8. Those three steps are, 1) preprocessing of the input program: with regard to the results obtained from the related researchers such as [33,34], a considerable number of program instructions does not have any effect on the program output results. This step includes a method called program slicing [35], which eliminates some of those instructions that do not have an impact on the program output results. The first step improves the speed of proposed GA in the second step, 2) identifying the most vulnerable blocks: GA has been proposed to identify vulnerable blocks. GA takes the source code of the program as an input to find out the smallest subset of the basic blocks which are more vulnerable. The most vulnerable blocks are identified based on initial population, selection, crossover, mutation, evaluation, and replacement processes introduced in GA, as clearly explained in [16], and 3) strengthening the identified vulnerable blocks: based on the required level of reliability, most vulnerable blocks in the program are strengthened against errors [16], is shown in Figure 8.

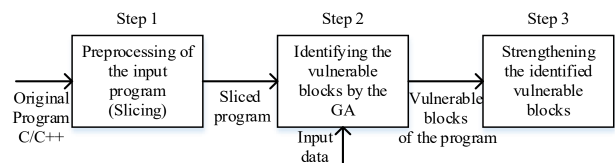


Figure 8. Representation of proposed method [16].

Strengths

Due to initial preprocessing and then selective vulnerable block duplication and comparison, the presented technique uses a less number of redundant instructions. Usage of a less number of redundant instructions decreases its performance and memory overhead. Performance and memory overhead presented in this technique are 24.3% and 60.3% [16].

Weaknesses

As other selective duplication techniques presented in this section, this technique considers only the most vulnerable blocks in the program for duplication. By duplicating only the vulnerable blocks in the program, most of the faults can be detected but not all. There is a possibility of undetected errors in the other normal blocks which lead to a reduction in fault coverage. Fault coverage achieved with this is technique is 82% [16].

Shoestring: Probabilistic soft error reliability

In the program, any instruction that can potentially influence global memory is considered as a high-value instruction [1]. In fact, if it consumes a wrong input, they are likely to produce outputs that result in user-visible corrupted results. In this

technique, high-value instructions are defined as the most vulnerable instructions and will have a huge impact on program output. Shoestring technique contributes in different issues for detecting the errors. Such issues are: 1) a transparent software solution for addressing soft errors, 2) a new reliability-aware compiler analysis, and 3) a selective instruction duplication that leverages compiler to identify and duplicate a small subset of vulnerable instructions [1]. Code duplication begins by selecting a single high-value instruction, from the set of all high-value instructions in the program. The selected single high-value instruction then proceeds to duplicate and then compare with comparison instruction. Duplication process is terminated when no more producers exist for duplication or the producer is already duplicated. Then the inserted comparison instructions are used for checking the errors. In Figure 9, the shaded parts represent the code duplication chains and the dashed circles indicate high-value instructions.

Strengths

Shoestring is a minimally invasive software solution, which results in very low overhead. Since duplicating and comparing only high-valued instructions, less number of redundant instructions are used which leads to decrease in overhead. Performance and memory overhead introduced in this technique are 20.16% and 40% [1], these are better than any other techniques in this field.

Weaknesses

Shoestring approach initially identifies the most unsafe instructions, as high-value instructions. Only duplicating and comparing the high-valued instructions, produce better overhead. At the same time, fault coverage is reduced because of possible undetected errors in the unduplicated instructions. Fault coverage achieved with this technique is 80.6 [1]%.

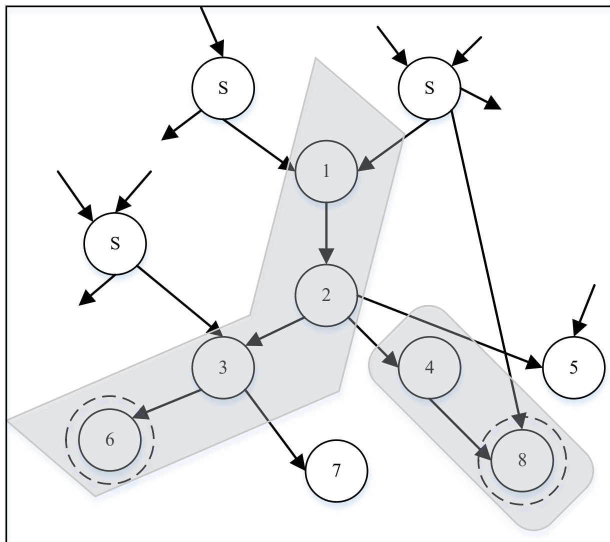


Figure 9. Example of data flow graph illustrating shoestrings code duplication chains [1].

4. Error Recovery

Error recovery techniques have been implemented to recover from the identified errors to keep systems in an error-free state with minimum overhead. Error recovery is generally based on the checkpointing concept [19–21,36]. Checkpoints are saved at regular intervals in the program based on the program execution behavior.

This section presents and reviews general checkpointing techniques for rollback error recovery and roll-forward error recovery. Strengths and weaknesses of rollback and roll-forward error recovery policies with checkpointing techniques are discussed.

4.1. Rollback Error Recovery

Rollback error recovery is one of the most used error recovery policies to recover the errors by using the checkpointing techniques in embedded systems. Bashiri, et al. propose a checkpointing technique for rollback error recovery. In rollback error recovery, in the case of an error, the processor state is restored to the error-free state with lower overhead [21]. In general, cost, performance, and memory overhead are primary factors for any error recovery technique.

The primary step for developing an error recovery technique is defining the correct error model. In [21], the proposed technique is based on control flow error model. During compilation time, the program is partitioned into basic blocks. The basic block is a set of instructions in a program without a jump instruction. Thereafter, an error detection mechanism needs to be added to the basic blocks presented in the program. Figure 10 shows the example of placing checkpoints in the control flow graph of the program. Usually, the checkpoint is stored in memory for rolling back the system with an immediate effect whenever an error is detected. Such a memory must be a fault-tolerant memory. The checkpoint contains the content of the registers, stack pointer and memory locations like stack region, constants, and variables [21]. For the considered benchmark programs such as bubble sort, matrix multiplication, and linked list copy, a checkpoint capturing is inserted to each of the basic blocks individually.

For example, a control flow graph is constructed with six blocks based on the program. Then checkpoints are added to the blocks based on the program execution order as shown in Figure 10. Since there will be a possibility of error occurrence before the first checkpoint location, it is mandatory to put a checkpoint at the beginning of the program. Remaining checkpoints are placed at the locations based on the program's vulnerability. In Figure 10, locations of the second and third checkpoints contain the vulnerable information. During the program execution time, whenever an error is detected, a detection mechanism informs the recovery routine and recovery routine recovers the error from the previously restored checkpoint. To evaluate the presented checkpointing technique, a pre-processor has been implemented that selects and adds the checkpoints to blocks [21].

Strengths

Bashiri, et.al, proposes a general checkpointing technique for rollback error recovery to recover from detected errors. The advantage of rollback error recovery is that if the error is detected, the processor state is restored into error free state without using the spare processor. The number of redundant instructions needed for rollback error recovery is very low. In the presented checkpointing technique for rollback error recovery, appeared memory overhead is low and also a cost efficient.

Weaknesses

In the presented checkpointing technique for rollback error recovery, whenever an error is detected, immediately the system

must rollback to its previous checkpoint of the corresponding program. Due to the fact that considerable time overhead is the main drawback of the rollback error recovery, as shown in Table 4. This technique is not fit for the typical time critical applications.

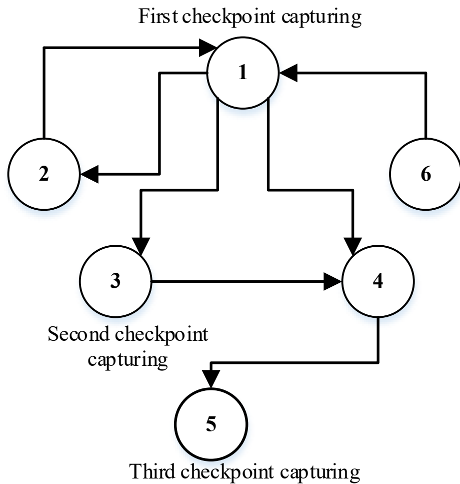


Figure 10. Periodic checkpointing representation in control flow graph.

4.2. Roll-Forward Error Recovery

Roll-forward error recovery is another error recovery policy to recover the errors. Roll-forward schemes are developed to increase the possibility that a given process completes within a given time. In that case, a couple of roll-forward schemes uses a spare processor for removing the rollback to save the time. At the same time, in time critical applications, redundancy is an important factor to consider because of cost, power, memory, and other factors. However, both schemes for roll-forward recovery with and without spare processor are discussed in this section.

Roll-forward recovery with dynamic replication checks (with spare processor)

The presented roll-forward recovery scheme uses dynamic replication checks to detect errors and is named RFR-RC (Roll-Forward Recovery with dynamic Replication Checks). This scheme is organized based on the isolated checkpoint intervals. For any isolated checkpoint intervals, a task is executed on two independent processors such as processor P1 and processor P2 as shown in Figure 11 [20]. In the presented scheme, at every checkpoint, the duplicated task records its state in the storage and the recorded state is forwarded to the checkpoint processor. Thereafter, at the end of the checkpoint interval, the checkpoint processor compares the two states from the processors. If the compared checkpoint states match with each other, the checkpoint will be committed and both the processors P1 and P2 continue their executions into the next checkpoint interval [20,21]. During the comparison, if any mismatch is detected, a validation step starts immediately. During the validation process, processors P1 and P2 continue their execution. At the same time, a spare processor is occupied to retry the last checkpoint interval using the previously committed checkpoint. Once the spare processor is ready with its process, the state is compared with the previous states of processors P1 and P2. The faulty processor among two processors such as P1 and P2 will be identified after this comparison. Then the identified faulty processor state will be made identical to that of the other processor. Now, both processors duplicating the task need to be in the correct state. As from the assumption [20] of single independent faults, a further validation process is not required.

Strengths

In the presented RFR-RC scheme, a spare processor is used to save time. With an extra processor, there is no need of rolling back to restore the system from error state. In RFR-RC, during the validation, the spare processor is used to identify the faulty processor and recovery action will be taken immediately. Appeared time overhead in RFR-RC scheme is decreased in comparison to the rollback scheme and RFR-BC scheme, as shown in Table 4.

Weaknesses

Because of using a spare processor to avoid the rollback, the cost is getting high. Memory overhead appeared in this technique is increased in comparison to the rollback recovery scheme.

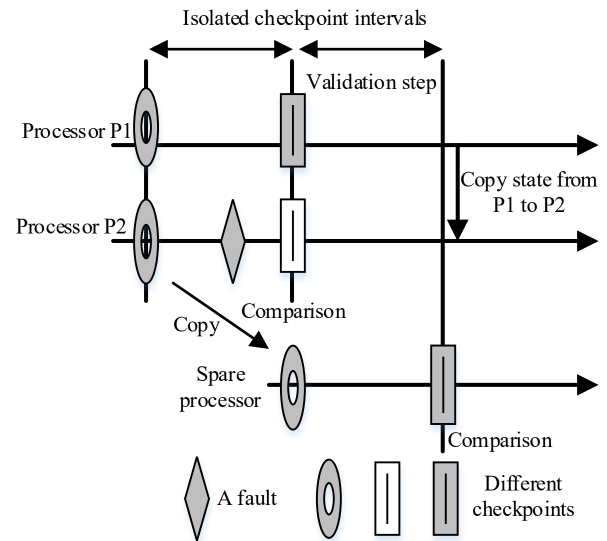


Figure 11. Roll-forward recovery in RFR-RC scheme [19,20].

Roll-forward recovery with behaviour-based checks (without spare processor)

In order to avoid the rollback, self-checks have been inserted to identify the faulty processors [20]. Such a self-detection methods are behaviour-based checks, such as control flow monitoring, detecting an illegal instruction, and memory protection. In [20,37], a new scheme has been proposed and implemented for roll-forward recovery named RFR-BC (Roll-Forward Recovery with Behavior based Checks). The proposed scheme uses a process pair approach to avoid the rollback to reduce the time. The intuitive idea presented in this scheme is, whenever an active task fails, the spare task becomes active to provide the necessary services [20]. The information sending through the active and spare task do not differ. However, information passing through the spare task need to be verified by the acceptance test before sending. Thereafter, the states of two processors (processor1 and processor 2) are verified at the end of a checkpoint interval to declare passing the test is committed, as shown in Figure 12.

Basically, acceptance test for sending the information validates a couple factors such as timing, coding, reasonableness, structural and diagnostic checks [20,38]. In the presented scheme, checkpointing is used for fault identification and roll-forward error

recovery. Whenever a faulty processor is located, then its state is made identical to the checkpoint state of the error-free processor. Because of this, both processors (processor1 and processor 2) will be in the correct state at the beginning of the next checkpoint interval. Figure 12 demonstrates the REF-BC scheme.

Strengths

RFR-BC scheme in roll-forward recovery does not need a spare processor as in RFR-RC to avoid rollback. In the RFR (Roll-Forward Recovery) schemes, the continuity of the executing program will be maintained so that the recovery delay will be removed [20,21]. The advantage of RFR-BC scheme is that time overhead is decreased in comparison to the rollback scheme, as shown in Table 4. It is also a more cost efficient solution compared to RFR-RC because of no spare processor.

Table 3. Results of the presented data error detection techniques from literature [1,7,12–17].

Error detection technique	Case studies used	Fault coverage (%)	Performance overhead (%)	Memory overhead (%)
EDDI	Insertion sort Quicksort Matrix mul. **(Average)**	99 98.2 99.2 **(98.8)**	113.90 89.3 111.1 **(104.7)**	200 200 200 **(200)**
ED^{4}I	Quicksort Matrix mul. Insertion sort **(Average)**	96.4 95.9 96.2 **(96.1)**	110 133 137 **(126.6)**	164 170 146 **(160)**
Software approach	Constant mod-ulus algorithm	100	244	362
CBD	Bubble sort Matrix mul. Quicksort **(Average)**	78.54 71.53 70.57 **(73.5)**	51 57 42 **(50)**	94 109 102 **(101.6)**
CPD	Differential equation solver	73.3	60	103
SEDSR	Bubble sort Matrix mul. Quicksort **(Average)**	95.01 93.95 95.59 **(94.85)**	112 121 104 **(112.3)**	127 146 131 **(134.6)**
Checking rules	Bubble sort Matrix mul. Dijkstra's **(Average)**	95.6 96 93.5 **(95)**	70 82 65 **(72.3)**	82 89 75 **(82)**
GA	Bubble sort Regression Quicksort **(Average)**	82 80 84 **(82)**	23 24 26 **(24.3)**	64 57 60 **(60.3)**
Shoestring	Vortex Crafty Gap **(Average)**	92 75 75 **(80.6)**	12 24 24.5 (20.16)	31 48 41 **(40)**

Table 4. Results of the presented error recovery techniques from literature [19– 21,36].

Method	Time overhead
Rollback recovery	31.1%
Roll-forward recovery (RFR-RC)	1.23%
Roll-forward recovery (RFR-BC)	2.08%

Weaknesses

In the RFR-BC, appointed self-check detection has an inaccurate error coverage and can't detect certain types of faults such as faults causing infinite looping. Time overhead with RFR- BC scheme is increased in comparison with the RFR-RC scheme for roll-forward recovery.

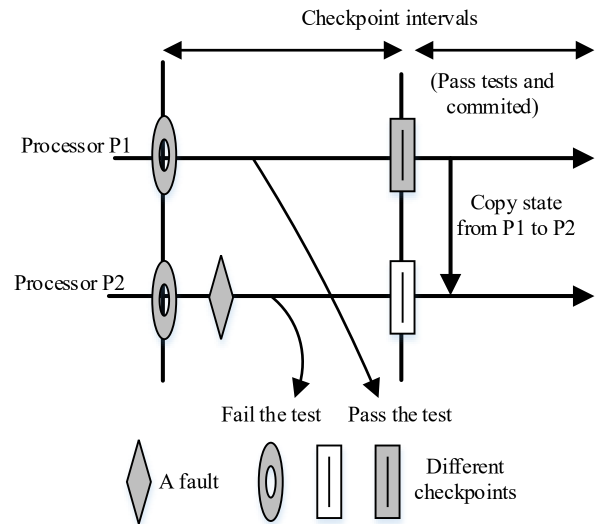


Figure 12. Roll-forward recovery in RFR-BC scheme [19,20].

Table 5. Strengths and weaknesses comparison of error detection methods.

Error detection technique	Strengths	Weaknesses
EDDI	The EDDI technique achieves high fault coverage by placing a comparison instruction after each MI and SI in the program.	Each of the original instruction is converted into three instructions, so performance and memory overhead is increased.
ED^{4}I	This technique achieves a high fault coverage when using the optimum value for *k*.	Good for either integers or floating point numbers, but not for both.
Software approach	Fault coverage achieved with this technique is very high because of duplicating the entire program.	More number of required instructions for duplication and comparison lead to increase in overhead.
CBD	Performance and memory overhead are decreased because they use less redundant instructions for duplication	This technique is a compiler and/or case study dependent and could just act as a full duplication technique.
CPD	Due to less number of instruc-tions required for duplication, performance and memory over-head is decreased.	In CPD, creating a data flow graph is much harder in assembly than C and C++.
SEDSR	In SEDSR, by placing a comparison instruction after writing to the classified final variables detect a lot of errors.	This technique is also a compiler and/or case study dependent and could just act as a full duplication.
Checking rules	The appeared performance and memory overhead in this technique are decreased in comparison to the EDDI, and ED^{4}I techniques.	Fault coverage achieved with this technique is reduced because of using less number of redundant instructions with the choice of checking rules.
GA	Because of selective vulnerable block duplication, the presented technique uses a less number of redundant instructions.	There is a possibility of undetected errors in the other normal blocks which lead to a reduction in fault coverage.
Shoestring	By duplicating and comparing only high-valued instructions, leads to decrease in overhead.	Fault coverage is reduced because of the possible undetected errors in the unduplicated instructions.

Table 6. Strengths and weaknesses comparison of error recovery methods.

Error recovery	Strengths	Weaknesses
Rollback error recovery	If the error is detected, the processor state is restored into error free state with out using spare processor.	Time overhead is the main drawback of the rollback error recovery, because of its rollback.
RFR-RC	Time overhead in RFR-RC scheme is decreased.	Usage of a spare processor to avoid the rollback, the cost is getting high.
RFR-BC	RFR-BC scheme in roll-forward recovery does not need a spare processor as in RFR-RC to avoid rollback.	Self-check detection has an inaccurate error coverage and can't detect certain types of faults.

5. Future Work

The strengths and weaknesses given in this paper for each technique were determined theoretically, by analyzing the technique and determining what data flow errors they detect, which they neglect and which overhead the techniques introduce.

To guide researchers and embedded systems engineers, we will perform an experimental comparison of the mentioned techniques. This experimental comparison will allow to evaluate the techniques on the same base: same hardware, same case studies, and same fault injection process. We'll perform this comparison both for data flow detection techniques and data flow recovery techniques. The outcome of the experimental comparison will allow other applicants of the techniques to quickly determine which existing technique is the best, in general or for their application.

Finally, we'll use the gathered data from the experimental comparison to develop a technique that can detect and recover from data flow errors, without introducing abnormal overhead.

6. Conclusions

This review paper lists and reviews various data flow error detection and recovery techniques existing in literature in the field of embedded systems. Each of the considered data error detection and correction technique has been discussed in terms of strengths and weaknesses. The discussion is summarized in Tables 5 and 6.

After thoroughly reviewing the strengths and weaknesses of error detection techniques, we have found that some of the techniques such as *ED^4I, EDDI and Software approach* are good for fault coverage but come with high overhead both in memory and performances. On the other hand, some other techniques such as *CBD, CPD, GA and Shoestring* are good for overhead but come with a reduction in fault coverage. At the same time, there exist a couple of techniques such as *SEDSR and Checking rules* which are good for fault coverage with satisfactory overhead.

As error recovery is concerned, based on the identified strengths and weaknesses from the methodology, we have found that *checkpointing technique for rollback error recovery* is better if memory overhead is the main concern. *Checkpointing technique for roll-forward error recovery schemes* are better if time overhead is the main concern.

Conflict of Interest

The authors declare no conflict of interest.

Acknowledgment

This work was funded by a technology transfer (TETRA) grant from the Flemish government in Belgium (VLAIO), under grant number IWT-150128 RELIM (Real-life immunity for embedded systems).

References

[1] Feng S, Gupta S, Ansari A, Mahlke S. Shoestring: probabilistic soft error reliability on the cheap: In: ACM SIGARCH Computer Architecture News.
[2] IEC 61508; Available from: http://www.iec.ch/functionalsafety/explained/.
[3] Li A, Hong B. Software implemented transient fault detection in space computer. Aerospace science and technology 2007;11(2):245–52.
[4] Thati VB, Vankeirsbilck J, Boydens J. Comparative study on data error detection techniques in embedded systems: In: 2016 XXV International Scientific Conference Electronics (ET), Sozopol, Bulgaria.
[5] S. Jagannathan, Z. Diggins, N. Mahatme, T. D. Loveless, B. L. Bhuva, S-J. Wen, R. Wong, and L. W. Massengill (ed.). Temperature dependence of soft error rate in flip-flop designs: IEEE; 2012.
[6] R. Baumann. Soft errors in advanced computer systems 2005;22(3):258–66.
[7] Oh N, Mitra S, McCluskey EJ. ED 4 I: error detection by diverse data and duplicated instructions. Computers, IEEE Transactions on 2002;51(2):180–99.
[8] Reis GA, Chang J, Vachharajani N, Rangan R, August DI. SWIFT: Software implemented fault tolerance: In: Proceedings of the international symposium on Code generation and optimization.
[9] Seyyed Amir Asghari, Okyaya Kaynak, and Hassan Taheri. An investigation into Soft Error Detection Efficiency at Operating System Level 2014.
[10] Dhiraj K. Pradhan, Nitin H. Vaidya: 24th International symposium on fault-tolerant computing Papers: FTCS-24; 1994.
[11] Chris Inacio. Software Fault Tolerance; Available from: https://users.ece.cmu.edu/~koopman/des_s99/sw_fault_tolerance/.
[12] Abdi A, Asghari SA, Pourmozaffari S, Taheri H, Pedram H. An Optimum Instruction Level Method for Soft Error Detection. International Review on Computers and Software 2012;7(2).
[13] Oh N, Shirvani PP, McCluskey EJ. Error detection by duplicated instructions in super-scalar processors. Reliability, IEEE Transactions on 2002;51(1):63–75.
[14] Abdi A, Asghari SA, Pourmozaffari S, Taheri H, Pedram H. An Effective Software Implemented Data Error Detection Method in Real Time Systems: In: Advances in Computer Science, Engineering & Applications: Springer; 2012, p. 919–26.
[15] Chielle E, Kastensmidt FL, Cuenca-Asensi S. Overhead Reduction in Data-Flow Software-Based Fault Tolerance Techniques: In: FPGAs and Parallel Architectures for Aerospace Applications: Springer; 2016, p. 279–91.
[16] Arasteh B, Bouyer A, Pirahesh S. An efficient vulnerability-driven method for hardening a program against soft-error using genetic algorithm. Computers & Electrical Engineering 2015;48:25–43.
[17] Asghari SA, Abdi A, Taheri H, Pedram H, Pourmozaffari S, others. SEDSR: soft error detection using software redundancy. Journal of Software Engineering and Applications 2012;5(09):664.
[18] Oh N, McCluskey EJ. Error detection by selective procedure call duplication for low energy consumption. Reliability, IEEE Transactions on 2002;51(4):392–402.
[19] Pradhan DK, Vaidya NH. Roll-Forward Checkpointing Scheme: A Novel Fault-Tolerant Architecture. IEEE Trans. Comput. 1994;43(10):1163–74.
[20] Jie Xu B. Roll-forward error recovery in embedded real-time systems: 1996 international conference on parallel and distributed systems; 1996.
[21] Mohsen Bashiri, Seyed Ghassem Miremadi and Mahdi Fazeli (ed.). A Checkpointing Technique for Rollback Error Recovery in Embedded Systems: IEEE Xplore; 2006.
[22] Nicolescu B, Savaria Y, Velazco R. Software detection mechanisms providing full coverage against single bit-flip faults. Nuclear Science, IEEE Transactions on 2004;51(6):3510–8.
[23] Karnik T, Hazucha P. Characterization of soft errors caused by single event upsets in CMOS processes. IEEE Trans.Dependable and Secure Comput. 2004;1(2):128–43.
[24] Arasteh B, Miremadi SG, Rahmani AM. Developing Inherently Resilient Software Against Soft-Errors Based on Algorithm Level Inherent Features. J Electron Test 2014;30(2):193–212.
[25] M. Li, P. Ramachandran, S. Sahoo, S. Adve, V. Adve, and Y. Zhou (ed.). SWAT: An Error Resilient System; 2008.
[26] Vankeirsbilck J, Thati VB, Hallez H, Boydens J. Inter-block jump detection techniques: A study: In: 2016 XXV International Scientific Conference Electronics (ET), Sozopol, Bulgaria.
[27] Oh N, Shirvani PP, McCluskey EJ. Control-flow checking by software signatures. Reliability, IEEE Transactions on 2002;51(1):111–22.

[28] Z. Alkhalifa, V.S.S. Nair, N. Krishnamurthy and J.A. Abraham. Design and evaluation of system-level checks for on-line control flow error detection 1999;10(6):627–41.
[29] O. Goloubeva, M. Rebaudengo, M. Sonza Reorda and M. Violante (ed.). Soft-error detection using control flow assertions: IEEE; 2003.
[30] Mei-Chen Hsueh, Tsai TK, Iyer RK. Fault injection techniques and tools. Computer 1997;30(4):75–82.
[31] Nicolescu B, Velazco R. Detecting soft errors by a purely software approach: method, tools and experimental results: In: Embedded Software for SoC: Springer; 2003, p. 39–51.
[32] Rebaudengo M, Reorda MS, Torchiano M, Violante M. Soft-error detection through software fault-tolerance techniques: In: Defect and Fault Tolerance in VLSI Systems, 1999. DFT'99. International Symposium on.
[33] Cook JJ, Zilles C. A characterization of instruction-level error derating and its implications for error detection: In: 2008 IEEE International Conference on Dependable Systems and Networks With FTCS and DCC (DSN), Anchorage, AK.
[34] Wang N, Fertig M, Sanjay Patel. Y-branches: when you come to a fork in the road, take it: In: 12th International Conference on Parallel Architectures and Compilation Techniques. PACT 2003, New Orleans, LA, USA, 27 Sept.-1 Oct. 2003.
[35] Weiser M. Program slicing: In: Proceedings of the 5th international conference on Software engineering.
[36] Dhiraj K. Pradhan, Nitin H. Vaidya. Roll-forward and rollback recovery: performance-reliability trade-off 1994;43(10).
[37] Silva JG, Silva LM, Madeira H, Bernardino J. A fault-tolerant mechanism for simple controllers: In: Goos G, Hartmanis J, Leeuwen J, Echtle K, Hammer D, Powell D, editors. Dependable Computing — EDCC-1. Volume 852. Berlin, Heidelberg: Springer Berlin Heidelberg; 1994, p. 39–55.
[38] Fault Tolerance: Principles and Practice: Springer-Verlag New York Inc; 2013.