CDFEDT—Comparison of Data Flow Error Detection Techniques in Embedded Systems: an Empirical Study

Venu Babu Thati
Dept. of Computer Science, M-Group,
KU Leuven Bruges Campus
Brugge, West Flanders, Belgium
Venubabu.Thati@kuleuven.be

Jens Vankeirsbilck
Dept. of Computer Science, M-Group,
KU Leuven Bruges Campus
Brugge, West Flanders, Belgium
Jens.Vankeirsbilck@kuleuven.be

Niels Penneman
R&D Department, Televic Healthcare
Izegem, West Flanders, Belgium
N.Penneman@televic.com

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

Jeroen Boydens
Dept. of Computer Science, M-Group,
KU Leuven Bruges Campus
Brugge, West Flanders, Belgium
Jeroen.Boydens@kuleuven.be

ABSTRACT

Embedded systems used in harsh environments are susceptible to bit-flips, which can cause data flow errors. In order to increase the reliability of embedded systems, numerous data flow error detection techniques have already been developed. It is, however, difficult to identify the best technique to apply, due to differences in the way they are evaluated in current literature.

This paper presents an empirical comparative study of seven existing techniques. We measured fault coverage, execution time overhead, and code size overhead. We conclude that soft error detection using software redundancy (SEDSR) and error detection by duplicated instructions (EDDI) have a better trade-off between fault coverage and overheads than software-implemented fault tolerance (SWIFT), critical block duplication (CBD), and overhead reduction (VAR3+). Error detection by diverse data and duplicated instructions (ED^4I or EDDDDI) and software approach (SA) had better fault coverage at the expense of execution time and code size usage.

CCS CONCEPTS

• **Computer systems organization** → **Embedded systems**; *Software reliability*; *Software fault tolerance*; • **General and reference** → *Reliability*; *Empirical experimental study*;

KEYWORDS

Data flow error detection, Fault coverage, Bit-flips, Comparison, Fault tolerance

ACM Reference Format:

Venu Babu Thati, Jens Vankeirsbilck, Niels Penneman, Davy Pissoort, and Jeroen Boydens. 2018. CDFEDT—Comparison of Data Flow Error Detection Techniques in Embedded Systems: an Empirical Study. In *ARES 2018: International Conference on Availability, Reliability and Security, August 27–30, 2018, Hamburg, Germany*. ACM, New York, NY, USA, 9 pages. <https://doi.org/10.1145/3230833.3230854>

1 INTRODUCTION

Advances in embedded systems are resulting in them becoming more susceptible to bit-flips, which are typically transient. This is mainly due to a reduction in the size of components and increasing complexity of the latest computer architectures [7]. Such bit-flips are caused by spontaneous changes in environments such as temperature fluctuations and electromagnetic interference [14, 15, 26]. Those bit-flips in embedded systems can cause irregular behavior and degrade the reliability and integrity of systems [18].

In embedded systems, bit-flips or soft errors do not consistently occur, unlike design or production errors. However, they are categorized as systematic errors as they will reappear in an identical manner under similar circumstances [21, 25]. According to the findings of recent studies, bit-flips are one of the immediate sources of failure in embedded systems [6]. Since there are more systems in use and the environment becomes harsher, a system crash or failure is more likely to occur. If we look from the software perspective, the effects of bit-flips can be divided into two categories; data flow and control flow errors.

Data flow errors lead to the corruption of data values in the program, causing a wrong intermediate value and thus final output value. In contrast, control flow errors lead to a jump (inter- and intra-block) in the program execution order, causing a system to hang or crash [9, 11, 19, 23, 28]. To improve the reliability and integrity of systems, fault tolerance should be included. Generally, fault tolerance can be achieved by techniques based on software or hardware redundancy. Techniques based on hardware redundancy provide good fault coverage at the expense of cost overhead. The most commonly used technique based on hardware redundancy for error detection is N parallel modules [2, 20, 24].

To reduce the overhead and cost, alternative techniques based on software redundancy need to be considered. When implemented

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.

ARES 2018, August 27–30, 2018, Hamburg, Germany

© 2018 Association for Computing Machinery.

ACM ISBN 978-1-4503-6448-5/18/08...\$15.00

<https://doi.org/10.1145/3230833.3230854>

in software, it is a one-time engineering cost at the start of the project, but overall a lower unit price can be achieved. Such techniques detect errors by introducing extra code and do not need to change the hardware [10]. These techniques require extra execution time and memory due to the execution of the extra code [29]. Software redundancy techniques offer high flexibility and cost effective solutions. This paper focuses on data flow errors. To counter data flow errors, numerous techniques have already been used for several years [2, 3, 7, 10, 16, 18, 22]. Some examples of these techniques: error detection by diverse data and duplicated instructions (ED^4I or EDDDDI) [18], error detection by duplicated instructions (EDDI) [20], and soft error detection using software redundancy (SEDSR) [7].

Although the results of the existing techniques are promising, it is difficult to identify the best technique based on the provided results in existing literature. These techniques were studied on different case studies which were executed on different hardware. Moreover, most techniques were validated using different fault injection processes. Therefore, they cannot be compared directly based on previous research and publications. To identify the best technique to apply, we conducted an empirical comparative study between the seven existing techniques: ED^4I [18], EDDI [20], software approach (SA) [16], software-implemented fault tolerance (SWIFT) [21], checking rules (VAR3+) [10], critical block duplication (CBD) [2], and SEDSR [7]. The main reason to select these techniques for our empirical study is; they are the most used techniques in the literature to perform comparisons with the newly developed techniques for evaluation [2, 3, 6, 21]. For our comparison, we implemented the seven techniques for the same case studies that executed on the same hardware and we validated them using one fault injection process. From experiments, we measured the techniques fault coverage, execution time overhead and code size overhead. Finally, we used these criteria to compare the seven techniques with one another. The best technique always had a better trade-off between fault coverage and overheads.

The remainder of this paper is organized as follows. Section 2 discusses about the related work. Section 3 gives a short overview of the seven considered data flow error detection techniques. Section 4 describes the experimental setup. Section 5 presents the results. Section 6 discusses a strategy for result comparison. Section 7 provides conclusions and our future work plans are given in Section 8.

2 RELATED WORK

As mentioned in the previous section, numerous data flow error detection techniques exist in the literature to address the data flow errors. Despite the fact that numerous techniques to detect data flow errors exist, identifying the efficient and best one among them is quite difficult based on the provided result in the literature. Therefore, in our previous research analysis and literature review, we performed a theoretical comparative study between numerous data error detection techniques to identify the best one [24, 25].

In addition, multiple authors have also published comparisons of fault tolerant techniques on theoretical basis [6, 12]. However, such theoretical comparisons are not so effective to identify the best technique in an efficient way. This is mainly because each technique follows its own validation mechanisms and use different hardware

for execution. To identify the best technique in a more reliable and efficient way, there is a need for experimental validation and comparison of results. Consequently, we performed experimental comparative study in this paper. That means, we reimplemented them by using same case studies, same hardware and same fault injection mechanism for all considered techniques. In this way, it is possible to validate and identify the best technique efficiently.

Duplication and then comparison are the basic software-based mechanisms involved in most data flow error detection techniques to detect errors [2, 3, 7, 10, 18, 20, 21, 24, 25]. Moreover, such mechanisms can be further divided into the following two categories: full code duplication and selective code duplication. This division is based on duplication mechanism involved in the corresponding techniques.

2.1 Full Code Duplication

In full code duplication techniques, the original instructions are duplicated and the results of the original and duplicated instructions are compared. A mismatch indicates a data flow error. Duplicating the entire code base results in a good/high fault coverage, but does impose higher code size and execution time overhead [16, 18, 20]. From our seven selected techniques, ED^4I , EDDI and SA are full code duplication techniques.

2.2 Selective Code Duplication

In selective code duplication techniques, only a part of the code is duplicated and their results are compared, resulting in reduced fault coverage with improvement in code size and execution time overhead [2, 7, 10, 21]. Among the selected techniques, SWIFT, VAR3+, CBD, and SEDSR are selective code duplication techniques. Most of the proposed selective code duplication techniques were based on a wide variety of possibilities to identify the error-prone blocks in a control flow graph (CFG). The CFG is a representation of the program and its flow, in which the program is divided into a number of basic blocks and edges. A basic block is a sequence of consecutive instructions with exactly one entry and one exit point [5, 27]. An edge is intentional path between the basic blocks based on branch and jump instructions.

In addition, an example of assembly-level instruction duplication and comparison is shown in Figure 1. In the figure, an unprotected basic block is shown on the left-hand side. The right-hand side shows the basic block protected by duplication and comparison. The duplicated instructions also need registers containing the same data as in the original instruction. These duplicated registers are indicated with an ', meaning that $R1'$, $R2'$, $R3'$ and $R4'$ are the duplicate of registers $R1$, $R2$, $R3$ and $R4$. The comparison instruction (CMP) compares the original and duplicated registers with each other and reports an error in case of a mismatch.

3 DATA ERROR DETECTION TECHNIQUES

This section provides an overview of all data error detection techniques that were selected in this study. We explain how duplication and comparison mechanisms used in each technique.

Figure 1: An example of assembly-level code duplication.

3.1 ED⁴I

ED⁴I or EDDDDI is a full code duplication mechanism. In *ED⁴I*, all instructions in the basic blocks are duplicated [18]. Comparison instructions have to be placed after each original and duplicated instruction in each basic block to compare their result. In this technique, transformed/duplicated data was changed by multiplying with factor k . In our implementation, we considered factor $k = 1$. The considered factor matches better with other duplication and comparison mechanisms selected for this study.

3.2 EDDI

EDDI is another full code duplication mechanism with selective comparison. Therefore, all instructions in the basic blocks are duplicated. In EDDI, comparison instructions have to be placed before store and branch instructions [20]. In this way, errors are detected.

3.3 SA

SA is also a full code duplication mechanism with selective comparison. In this technique, all instructions in the basic blocks are duplicated as in *ED⁴I* and EDDI. Comparison instructions have to be placed after the last original and duplicated instructions in each basic block to compare their results [16].

3.4 SWIFT

SWIFT is a selective code duplication mechanism based on the instructions. It is an improved version of EDDI. In this technique, all instructions in the basic blocks, except store instructions, are duplicated [21]. Comparison instructions have to be placed before store and branch instructions to compare the results.

3.5 VAR3+

VAR3+ is another selective code duplication mechanism based on the instructions. In this technique, all instructions in the basic blocks, except branch and store instructions, are duplicated. Comparison instructions have to be placed before load, store, and branch instructions to compare the results [10].

3.6 CBD

CBD is a selective code duplication mechanism based on the basic blocks. In this technique, critical blocks have to be identified in the CFG. A block that has the highest number of fan-outs in the CFG is considered as the critical block [2]. Instructions in the identified critical blocks are duplicated and then results from original and duplicated instructions are compared with a comparison instruction. If any mismatch between the result is detected, an error is reported.

3.7 SEDSR

SEDSR is an extended version of CBD. In this technique, critical blocks have to be identified in the CFG. A block that has the highest number of fan-outs in the CFG is considered as the critical block. In SEDSR, comparison instructions have to be placed after the last original and duplicated instructions in each identified critical block to compare the results [7]. In this way, errors are detected.

All selected and discussed data flow error detection techniques are differentiated by their duplication and comparison mechanisms. Table 1 shows an example using a part of code hardened by the seven considered data flow error detection techniques for the ARM Cortex - M3 processor. The unhardened/original code is presented as normal text, while instructions inserted by the duplication mechanism are in *italic* and the comparison instructions are in **bold**. Furthermore, as mentioned in Table 1, add, ldr, str, mov, sub and cmp, indicates the assembly-level instructions of addition, load, store, move, subtraction and comparison operations, respectively. Likewise, ble, represents a branch instruction which is less or equal to a condition, whereas bne represents a branch instruction which is not equal to a condition.

In general, full code duplication techniques provide good fault coverage with extra imposed execution time and code size overheads. On the other hand, selective code duplication techniques provide an improvement in overheads with a reduction in fault coverage.

4 EXPERIMENTAL SETUP

This section describes case studies, hardware, and fault injection used in this study to perform experiments. In the end, we discuss which criteria are used to evaluate the seven implemented data error detection techniques.

4.1 Case Studies

To validate each of the selected data flow error detection technique, we implemented them for seven popular case studies in assembly language for an ARM Cortex-M3 processor. Those are an implementation of bubble sort (BS), quicksort (QS), matrix multiplication (MM), bit count (BC), cyclic redundancy check (CRC), insertion sort (IS), and selection sort (SS) algorithms. Next to being highly used as validation case studies in the published literature of the selected techniques, these case studies also have varying complexity and CFGs [27]. This makes them highly suitable for our experimental comparison.

4.2 Hardware

The selected case studies were executed on a NXP LPC1768, an ARM Cortex-M3 driven microcontroller executing at 96 MHz. It includes 512-kB FLASH and 32-kB RAM with interfaces for different communication buses. The ARM Cortex-M3 was selected because it is an industry leading 32-bit processor used in various embedded application domains [1].

4.3 Fault Injection Process and Tool

To validate the seven implemented data error detection techniques, bit-flips need to occur on the target system. To make data flow

Table 1: Example of Implemented Data Error Detection Techniques for ARM Cortex-M3

	Unhardened code	<i>ED⁴I</i>	EDDI	SA	SWIFT	VAR3+	CBD	SEDSR
1	add r1, r3	add r1, r3	add r1, r3	add r1, r3	add r1, r3	add r1, r3	add r1, r3	add r1, r3
2		<i>add r1', r3'</i>	<i>add r1', r3'</i>	<i>add r1', r3'</i>	<i>add r1', r3'</i>	<i>add r1', r3'</i>	<i>add r1', r3'</i>	<i>add r1', r3'</i>
3		cmp r1, r1'					cmp r1, r1'	
4		bne error					bne error	
5	ldr r2, [r3]	ldr r2, [r3] <i>ldr r2', [r3']</i> cmp r2, r2' bne error	ldr r2, [r3] <i>ldr r2', [r3']</i>	ldr r2, [r3] <i>ldr r2', [r3']</i>	ldr r2, [r3] <i>ldr r2', [r3']</i>	ldr r2, [r3] <i>ldr r2', [r3']</i>	ldr r2, [r3] <i>ldr r2', [r3']</i> cmp r2, r2' bne error	ldr r2, [r3] <i>ldr r2', [r3']</i>
6								
7								
8								
9								
10								
11	str r1, [r2]	str r1, [r2] <i>str r1', [r2']</i> cmp r1, r1' bne error	str r1, [r2] <i>str r1', [r2']</i>	str r1, [r2] <i>str r1', [r2']</i>	str r1, [r2] <i>str r1', [r2']</i>	str r1, [r2] <i>str r1', [r2']</i>	str r1, [r2] <i>str r1', [r2']</i> cmp r1, r1' bne error	str r1, [r2] <i>str r1', [r2']</i>
12								
13								
14								
15								
16								
17	mov r2, r0	mov r2, r0	mov r2, r0	mov r2, r0	mov r2, r0	mov r2, r0	mov r2, r0	mov r2, r0
18		<i>mov r2', r0'</i>	<i>mov r2', r0'</i>	<i>mov r2', r0'</i>	<i>mov r2', r0'</i>	<i>mov r2', r0'</i>	<i>mov r2', r0'</i>	<i>mov r2', r0'</i>
19		cmp r2, r2'					cmp r2, r2'	
20		bne error					bne error	
21	sub r0, r1	sub r0, r1	sub r0, r1	sub r0, r1	sub r0, r1	sub r0, r1	sub r0, r1	sub r0, r1
22		<i>sub r0', r1'</i>	<i>sub r0', r1'</i>	<i>sub r0', r1'</i>	<i>sub r0', r1'</i>	<i>sub r0', r1'</i>	<i>sub r0', r1'</i>	<i>sub r0', r1'</i>
23		cmp r0, r0'					cmp r0, r0'	
24		bne error					bne error	
25	cmp r3, r2	cmp r3, r2 bne error cmp r2, r2' bne error	cmp r3, r2 bne error cmp r2, r2' bne error	cmp r3, r2 bne error cmp r2, r2' bne error	cmp r3, r2 bne error cmp r2, r2' bne error	cmp r3, r2 bne error cmp r2, r2' bne error	cmp r3, r2 bne error cmp r2, r2' bne error	cmp r3, r2 bne error cmp r2, r2' bne error
26								
27								
28								
29								
30								
31								
32								
33								
34								
35	ble loop	ble loop	ble loop	ble loop	ble loop	ble loop	ble loop	ble loop

errors occur, bit-flips can be injected in the CPU registers. However, most systems already have some ECC countermeasures in place for their memory. Therefore we opted to only inject bit-flips in the CPU registers. To deterministically and directly inject data flow errors into the processor registers, the instruction aware injection mechanism was used. The instruction aware injection was proposed by Heinig et al. [13] to inject data flow errors. The fault injection process per bit-flip according to the instruction aware injection process consists of the following steps:

- 1) Halt target.
- 2) Read instruction to be executed.
- 3) Determine registers.
- 4) Pick a register to corrupt.

- 5) Inject a bit-flip.
- 6) Resume target.

The instruction aware injection process was integrated into our in-house developed software-implemented fault injection (SWIFI) tool. The advantage of the SWIFI tool over the other fault injection mechanisms in [4, 8] is; it deterministically injects fault with minor randomness, leading to a very good fault injection rate in the target system. The SWIFI tool focuses on the ARM Cortex-M processor family [1, 17]. The architecture of the SWIFI tool is shown in Figure 2. The external host PC is connected to the Cortex microcontroller software interface standard debug access port (CMSIS-DAP) interface of the processor to inject bit-flips into the target hardware. The CMSIS-DAP interface allows us to remotely manipulate data

Figure 2: Architecture of the SWIFI tool.

present in the CPU registers using the on-chip debugger. Next, 3200 data flow errors per case study were injected for validation.

Based on the effects of injected faults, obtained results are analyzed with the following classification:

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 duplication and comparison instructions used for data flow error detection.

Undetected Fault (UDF): The injected fault is not detected and it changes the output of the program.

4.4 Criteria

In this section, we discuss the three criteria measured for each implemented technique: fault coverage (FC), execution time overhead (ETO), and code size overhead (CSO). The fault coverage is measured as all detected faults. In other words, this is the sum of the Correct Result, Hard Fault, and Detected Fault. However, fault coverage does not come for free, as duplication and comparison mechanisms introduce both execution time and code size overhead. Execution time overhead indicates the extra time it takes to execute the protected code. Whereas, code size overhead indicates the increase in the size of the protected code. Definitions for the fault coverage, execution time overhead, and code size overhead are provided in Equations (1), (2), and (3).

$$FC = 1 - \frac{\text{Undetected faults}}{\text{Total injected faults}} \quad (1)$$

$$ETO = \frac{\text{Protected execution time} - \text{Unprotected execution time}}{\text{Unprotected execution time}} \quad (2)$$

$$CSO = \frac{\text{Protected code size} - \text{Unprotected code size}}{\text{Unprotected code size}} \quad (3)$$

5 RESULTS

This section discusses the results of our experiments. Figure 3 shows the results obtained from the fault injection experiments divided into the four fault categories. Figures 4 and 5 present execution time overhead and code size overhead, respectively.

Analyzing Figure 3, the three full code duplication mechanisms such as EDDDDI, EDDI, and SA show good percentage of Detected Fault ratio for most case studies. Among them, EDDDDI showed

the highest percentage of Detected Fault ratio in most of the case studies. This is because in EDDDDI all instructions present in the basic blocks are duplicated and compared, while in EDDI and SA all instructions present in the basic blocks are duplicated but only selective instructions result are compared. On the other side, percentage of Detected Fault ratio of the selective code duplication techniques such as SWIFT, VAR3+, CBD, and SEDSR is comparatively lower. This is because only a selected part of the code was duplicated. Among them, CBD showed better Detected Fault ratio because of protecting all instructions present in the critical blocks, while SWIFT, VAR3+, and SEDSR techniques show less percentage of Detected Fault ratio. Duplication and comparison mechanisms present in SWIFT and VAR3+ were based on the instructions and they protect fewer instructions which leads to a reduction in their Detected Faults. SEDSR shows a lower Detected Faults ratio in comparison to all other considered techniques. This is because of duplicating and comparing only selective instructions.

Figures 4 and 5 show execution time and code size overhead for the different case studies and different techniques. In Figure 4, the techniques SEDSR and SWIFT had less execution time overhead for most case studies. This was due to a low number of required additional instructions for duplication and comparison mechanisms involved in those techniques. Although being a selective duplication technique, CBD imposes a high overhead, matching that of full duplication techniques. This is because most of the basic blocks in the CFGs of the case studies were identified as critical blocks. Later, to reduce the overhead present in CBD, same authors who proposed CBD, extended it to SEDSR. As expected, full code duplication techniques show high execution time overhead. Among them, EDDDDI had a very high execution time overhead because of duplicating all instructions and comparing their results.

Code size overheads are given in Figure 5. They are calculated based on the number of duplicate and comparison instructions added to the original instructions. In Figure 5, the techniques SEDSR and VAR3+ indicates less code size overhead for most case studies. According to the duplication and comparison mechanisms involved in SEDSR and VAR3+, the need of less additional instructions, leads to low overheads. Similarly, as in execution time, EDDDDI had a very high code size overhead because of the additional requirement of instructions for its duplication and comparison mechanisms.

6 COMPARISON

This section discusses a strategy for result comparison. In order to compare different data error detection techniques with each other, Asghari et al. proposed the Evaluation Factor (EF) [7]. The EF considers the fault coverage and imposed overheads of the techniques and calculates the made trade-off as shown in Equation (4). However, it prefers methods in which one of the two overheads is very high and the other is very low. This is because the product term is usually minimized when one factor is small and another factor is high given that the total of the factors is fixed.

To overcome this problem, we adapted the EF to the Gauging Factor (GF). In our GF, we replaced the product with a weighted sum as shown in Equation (5). We opted to use a sum, to avoid one overhead dominating the other too much, as is the case with the product in EF. Using a weighted sum allows to indicate importance

Figure 3: Fault injection results of implemented techniques.

Figure 4: Execution time overhead.

Figure 5: Code size overhead.

to the considered overheads. The higher the GF, the better the trade-off between fault coverage and overheads. In Equation (5), P is the variable parameter.

$$EF = \frac{FC}{ETO \times CSO} \quad (4)$$

$$GF = \frac{FC}{(P \times ETO) + ((1 - P) \times CSO)} \quad (5)$$

To better explain the flaw of EF in our point of view, we calculated both the EF and GF for two example techniques. Each example technique contains the three criteria; FC, ETO, and CSO which are given in the bullet points below. As it can be seen, we assume the same FC for both *example techniques* with different overheads. When we calculate the EF, it determines *example technique two* as having the best trade-off. This is due to its low CSO. This minimizes the product in the denominator, thus giving this technique a higher EF. However, if the user finds both overheads of equal importance, *example technique one* should be its choice. When we calculate the

Figure 6: Fault coverage with regard to execution time overhead.

Figure 7: Fault coverage with regard to code size overhead.

Figure 8: Fault coverage with regard to execution time and code size overhead.

GF, with $P = 0.5$, we get the wanted result. *Example technique one* shows to make a better trade-off, for our indicated importance.

- **Example technique 1:** FC = 80%, ETO = 50%, and CSO = 50%
- **Example technique 2:** FC = 80%, ETO = 120%, and CSO = 10%
- * **EF-Example technique 1:** EF = 3.2
- * **EF-Example technique 2:** EF = 6.6
- * **GF-Example technique 1:** GF = 1.6
- * **GF-Example technique 2:** GF = 1.2

The remainder of this section discusses three possible trade-off calculations. First, we calculate the GF when only execution time matters ($P = 1$). Next, we calculate the GF when only the code size overhead is of concern ($P = 0$). Finally, we calculate the GF with an

equal importance for both overhead ($P = 0.5$). Figures 6, 7, and 8 present fault coverage with regard to execution time overhead, code size overhead, and both execution time and code size overhead.

FC regard to ETO

First, we calculate the trade-off between FC and ETO for the selected techniques. This means that GF is calculated with $P = 1$. In this case, the calculated GF values are shown in Figure 6. EDDI and SWIFT show to be case study independent, by having the highest GF values for most case studies. This is due to their high FC and low ETO. The other techniques are indicated as techniques making a decent trade-off, but being more case study dependent. The CBD

technique, for instance, shows to make the best trade-off for the SS case study but has the worst trade-off of all techniques for the IS case study. Similar conclusions can be drawn for SEDSR, VAR3+ and SA. EDDDDI often has the lowest trade-off and is therefore considered as the least optimum technique in this comparison.

FC regard to CSO

In this part, we calculate the trade-off between FC and CSO for each technique, thus using $P = 0$. As shown in Figure 7, most techniques show to be case study independent and have a high GF for all case studies. The SEDSR technique makes the best trade-offs for most case studies, with EDDDDI again indicated as the least optimum technique in the comparison. This result was to be expected, as Figure 5 indicates the same trend.

FC regard to ETO and CSO

Finally, we calculate the trade-off taking both measured overheads into account. The calculated GF values, with $P = 0.5$, are shown in Figure 8. As can be seen, EDDI and SEDSR show to make the better trade-offs consistently for most case studies. SA, SWIFT, VAR3+ and CBD indicate to make a comparable trade-off. EDDDDI again ended up having a poor trade-off due to the high execution time and code size overhead.

7 CONCLUSION

In this paper, we presented an empirical comparative study of seven data flow error detection techniques. We considered EDDDDI, EDDI, SA, SWIFT, VAR3+, CBD, and SEDSR for this study. We explained how each considered technique works based on their duplication and comparison mechanisms. The re-implementation of each technique was performed over seven most used case studies in existing literature on those seven techniques. To validate the implemented techniques, we performed fault injection experiments with SWIFT tool. The obtained results from the validation were: fault coverage, execution time overhead, and code size overhead.

To perform an empirical comparison, we proposed the use of GF in this paper. It calculates the trade-off between fault coverage and overheads. The experimental results and comparison show that, if execution time and code size overhead were not of interest, EDDDDI is the best technique to implement as it has a very good fault coverage. If execution time overhead is the main concern along with fault coverage, EDDI and SWIFT have a better trade-off. For applications where code size overhead is of main interest along with fault coverage, SEDSR had the best trade-off. If both execution time and code size overhead were of concern along with fault coverage, EDDI and SEDSR are the best techniques to implement. This empirical study is helpful for developers and researchers to identify the best technique to apply in numerous general purpose and real-time applications.

8 FUTURE WORK

Based on the lessons learned and ideas gathered during this empirical study, we are implementing a new data error detection technique. The aim of our new technique is to reduce the code size and execution time overhead with comparable fault coverage.

Further, in fault-tolerant embedded systems, error detection by itself is not enough to make them fault tolerant. It also needs error recovery. Thus, our future work will focus on this addition. In more detail, error recovery is generally based on the checkpointing concept. Checkpoints are saved at regular intervals to check for irregular behavior. Such techniques are further classified into rollback error recovery and roll-forward error recovery techniques.

REFERENCES

- [1] Online. "Cortex-M3 processor", ARM. (Online). Retrieved March 13, 2018 from <https://developer.arm.com/products/processors/cortex-m/cortex-m3>
- [2] A Abdi, SA Asghari, S Pourmzaffari, H Taheri, and H Pedram. 2012. An Optimum Instruction Level Method for Soft Error Detection. *International Review on Computers and Software* 7, 2 (2012).
- [3] Atena Abdi, Seyyed Amir Asghari, Saadat Pourmzaffari, Hassan Taheri, and Hossein Pedram. 2012. An Effective Software Implemented Data Error Detection Method in Real Time systems. In *Advances in Computer Science, Engineering & Applications*. Springer, 919–926.
- [4] Joakim Aidemark, Jonny Vinter, Peter Folkesson, and Johan Karlsson. 2001. Goofi: Generic object-oriented fault injection tool. In *Dependable Systems and Networks, 2001. DSN 2001. International Conference on*. IEEE, 83–88.
- [5] Frances E Allen. 1970. Control flow analysis. In *ACM Sigplan Notices*, Vol. 5. ACM, 1–19.
- [6] Bahman Arasteh, Asgarali Bouyer, and Sajjad Pirahesh. 2015. An efficient vulnerability-driven method for hardening a program against soft-error using genetic algorithm. *Computers & Electrical Engineering* 48 (2015), 25–43.
- [7] Seyyed Amir Asghari, Atena Abdi, Hassan Taheri, Hossein Pedram, and Saadat Pourmzaffari. 2012. SEDSR: soft error detection using software redundancy. *Journal of Software Engineering and Applications* 5, 09 (2012), 664.
- [8] Seyyed Amir Asghari, Hossein Pedram, Hassan Taheri, and Mohammad Khademi. 2010. A new Background Debug mode based technique for fault injection in embedded systems. *International Review on Modelling and Simulations* 3, 3 (2010), 415–422.
- [9] Tahereh Boroomandnezhad and Mohammad Abdollahi Azgomi. 2013. An efficient control-flow checking technique for the detection of soft-errors in embedded software. *Computers & Electrical Engineering* 39, 4 (2013), 1320–1332.
- [10] Eduardo Chielle, Fernanda Lima Kastensmidt, and Sergio Cuenca-Asensi. 2016. Overhead Reduction in Data-Flow Software-Based Fault Tolerance Techniques. In *FPGAs and Parallel Architectures for Aerospace Applications*. Springer, 279–291.
- [11] Olga Golubeva, Maurizio Rebaudengo, M Sonza Reorda, and Massimo Violante. 2003. Soft-error detection using control flow assertions. In *Defect and Fault Tolerance in VLSI Systems, 2003. Proceedings. 18th IEEE International Symposium on*. IEEE, 581–588.
- [12] Olga Golubeva, Maurizio Rebaudengo, Matteo Sonza Reorda, and Massimo Violante. 2006. *Software-implemented hardware fault tolerance*. Springer Science & Business Media.
- [13] Andreas Heinig, Ingo Korb, Florian Schmoll, Peter Marwedel, and Michael Engel. 2013. Fast and Low-Cost Instruction-Aware Fault Injection.. In *GI-Jahrestagung*. 2548–2561.
- [14] S Jagannathan, Z Diggins, N Mahatme, TD Loveless, BL Bhuva, SJ Wen, R Wong, and LW Massengill. 2012. Temperature dependence of soft error rate in flip-flop designs. In *Reliability Physics Symposium (IRPS), 2012 IEEE International*. IEEE, SE–2.
- [15] Tanay Karnik and Peter Hazucha. 2004. Characterization of soft errors caused by single event upsets in CMOS processes. *IEEE Transactions on Dependable and Secure Computing* 1, 2 (2004), 128–143.
- [16] Bogdan Nicolescu and Raoul Velazco. 2003. Detecting soft errors by a purely software approach: method, tools and experimental results. In *Embedded Software for SoC*. Springer, 39–51.
- [17] Kyle Norman. 2007. Integrating Web Servers in Embedded Applications. *Luminary Micro Applications Engineer, Luminary MicroWild Basin, Suite 350* (2007).
- [18] Nahmsuk Oh, Subhasish Mitra, and Edward J. McCluskey. 2002. ED4I: Error detection by diverse data and duplicated instructions. In *IEEE Transactions on Computers* 51 (2002), 180.
- [19] Nahmsuk Oh, Philip P Shirvani, and Edward J McCluskey. 2002. Control-flow checking by software signatures. *IEEE transactions on Reliability* 51, 1 (2002), 111–122.
- [20] Nahmsuk Oh, Philip P Shirvani, and Edward J McCluskey. 2002. Error detection by duplicated instructions in super-scalar processors. *IEEE Transactions on Reliability* 51, 1 (2002), 63–75.
- [21] George A Reis, Jonathan Chang, Neil Vachharajani, Ram Rangan, and David I August. 2005. SWIFT: Software implemented fault tolerance. In *Proceedings of the international symposium on Code generation and optimization*. IEEE Computer Society, 243–254.

- [22] Alex Shye, Joseph Blomstedt, Tipp Moseley, Vijay Janapa Reddi, and Daniel A Connors. 2009. PLR: A software approach to transient fault tolerance for multicore architectures. *IEEE Transactions on Dependable and Secure Computing* 6, 2 (2009), 135–148.
- [23] Lanfang Tan, Ying Tan, and Jianjun Xu. 2013. CFEDR: Control-flow error detection and recovery using encoded signatures monitoring. In *Defect and Fault Tolerance in VLSI and Nanotechnology Systems (DFT), 2013 IEEE International Symposium on*. IEEE, 25–32.
- [24] Venu Babu Thati, Jens Vankeirsbilck, and Jeroen Boydens. 2016. Comparative study on data error detection techniques in embedded systems. In *Scientific Conference Electronics (ET), International*. IEEE, 1–4.
- [25] Venu Babu Thati, Jens Vankeirsbilck, Jeroen Boydens, and Davy Pissoort. 2017. Data Error Detection and Recovery in Embedded Systems: a Literature Review. *Advances in Science, Technology and Engineering Systems Journal* 2, 3 (2017), 623–633.
- [26] Jonas Van Waes, Jens Vankeirsbilck, Davy Pissoort, and Jeroen Boydens. 2017. Electromagnetic Interference in the Internet of Things: an automotive insight. In *Scientific Conference Electronics (ET), 2017 XXVI International*. IEEE, 1–4.
- [27] Jens Vankeirsbilck, Niels Penneman, Hans Hallez, and Jeroen Boydens. 2017. Random Additive Signature Monitoring for Control Flow Error Detection. *IEEE Transactions on Reliability* (2017).
- [28] Jens Vankeirsbilck, Venu Babu Thati, Hans Hallez, and Jeroen Boydens. 2016. Inter-block jump detection techniques: A study. In *Scientific Conference Electronics (ET), International*. IEEE, 1–4.
- [29] Tianwen Yao, Haifang Zhou, Minquan Fang, and Haibin Hu. 2013. Low power consumption scheduling based on software fault-tolerance. In *Natural Computation (ICNC), 2013 Ninth International Conference on*. IEEE, 1788–1793.