Journal of Software Engineering and Applications, **2012, 5, 664-670**
http://dx.doi.org/10.4236/jsea.2012.59078 Published Online September 2012 (http://www.SciRP.org/journal/jsea)

SEDSR: Soft Error Detection Using Software Redundancy

Seyyed Amir Asghari[1], Atena Abdi[1], Hassan Taheri[2], Hossein Pedram[1], Saadat Pourmozaffari[1]

[1]Computer Engineering and Information Technology Department, Amirkabir University of Technology, Tehran, Iran; [2]Electrical Engineering Department, Amirkabir University of Technology, Tehran, Iran.
Email: seyyed_asghari@aut.ac.ir, atena.abdi@aut.ac.ir, htaheri@aut.ac.ir, pedram@aut.ac.ir, saadat@aut.ac.ir

Received July 3rd, 2012; revised August 7th, 2012; accepted August 20th, 2012

ABSTRACT

This paper presents a new method for soft error detection using software redundancy (SEDSR) that is able to detect transient faults. Soft errors damage the control flow and data of programs and designers usually use hardware-based solutions to handle them. Software-based techniques for soft error detection force less cost and delay to systems and do not change their configuration. Therefore, these kinds of methods are appropriate alternatives for hardware-based techniques. SEDSR has two separate parts for data and control flow errors detection. Fault injection method is used to compare SEDSR with previous methods of this field based on the new parameter of "Evaluation Factor" that takes in account fault coverage, memory and performance overheads. These parameters are important in real time safety critical applications. Experimental results on SPEC2000 and some traditional benchmarks of this field show that SEDSR is much better than previous methods of this field. SEDSR's evaluation factor is about 50% better than other methods of this field. These results show its success in satisfaction of the existing tradeoff between fault coverage, performance and memory overheads.

Keywords: Soft Error Detection; Control Flow Errors; Data Errors; Evaluation Factor; Fault Injection

1. Introduction

It is proved that decreasing in feature size of integrated circuits and the increased complexity of computer architectures will lead to less reliable systems. The most important kinds of errors that threat computer systems and become more important with technology advances are soft errors. Unlike manufacturing and design faults, soft errors do not occur consistently and cannot be predicted which are also called transient faults. Recent researches show that soft errors damage control flow or data of a program. It is proved that about 33% to 77% of transient faults are converted to Control Flow Errors (CFE) and the remained percentage is converted to data errors [1,2]. The first and the most important step for making systems tolerant against transient faults is fault detection and success in this step can provide appropriate fault coverage for the system. For transient faults detection, some techniques are presented that can be categorized in two general classes of hardware and software-based. The methods based on hardware redundancy have better fault coverage but impose more cost and overhead to the system and may not satisfy users in general purpose applications because of system reconfiguration. On the other hand, software-based methods have less fault coverage and more delay; however, they have less cost and do not change the configuration of the system and are utilized in different applications due to their flexibility [3-11].

For control flow checking, the program code is divided into some basic blocks. Each block is consisted of instructions that are located among jumps. The errors that should be analyzed in control flow checking methods are classified into three general categories: 1) wrong occurred jumps intra a basic block; 2) wrong occurred jumps inter two basic blocks; and 3) wrong occurred jumps from a basic block to the unused space of the memory. The methods that are presented in this field, software or hardware-based, should be able to handle these kinds of errors as much as possible.

Hardware-based control flow checking methods monitor the behavior of the program by using a watchdog processor that compare the current flow of the program with its correct one [3]. Software-based methods on the other side assign a signature to each basic block and add some instructions to compare the amount of run time signatures with the pre-compile values which are saved in the memory.

As mentioned earlier, the remained percentage of soft errors damages the data variables that are used by applications. To manage this kind of errors, two approaches are usually utilized: hardware-based and the methods based on information redundancy. The former is utilizing

N parallel modules in order to do the operation and compare their results with each other. This technique has about 100(N − 1)% memory and performance overhead and its fault coverage is about 100% [4]. The later stores multiple copies of the program in the memory and compares their results with each other. Some solutions based on information redundancy can be found in [5,6]. These methods generally replicate the whole program and compare their results with each other. The comparison instructions are inserted in different places of the program in various methods.

The mentioned methods are widely used for control flow and data errors detection. These methods are evaluated and compared with each other by fault injection. As mentioned, by the technology progress and reduction of electronic equipment dimensions, delivering a method that is able to have good fault coverage by an acceptable overhead impose on memory and performance is very important. The parameters of memory consumption and execution time of a program is as important as fault coverage of a method nowadays. Other methods of this field do not meet these three parameters simultaneously. To meet all of the parameters, in this paper, a factor that is called "*Evaluation Factor*" is introduced and used for different methods comparison.

SEDSR that is presented in this paper first uses a control flow checking method that assigns a signature and some redundant instructions to each basic block. Then it duplicates a block which is called critical because of its effect on the other blocks. This method has acceptable fault coverage and its time/space overhead is less than 100% which is better than the previous presented methods of this field.

The reminder of this paper is organized as follows. In the second section of this paper, our motivation of presenting this method and the experimental conditions will be explained. The third section introduces the proposed method and the evaluation results and SEDSR comparison with other techniques is presented in the forth section.

2. Motivation

Usually, transient faults that occur in computer systems cause control flow checking or data errors. Data errors change the value of a variable in the program and control flow errors change the running flow of a program.

Soft errors which are mainly because of single event upsets (SEUs) are caused because of electromagnetic interference, power glitch or strikes on a chip. These kinds of errors are very serious and can interrupt the functionality of a program. So exploiting a method for soft error detection is needed in many applications which are using Commercial Off-The Shelf (COTS) equipment.

A control flow error occurrence in a program can be modeled by the following situations:

- **Branch deletion:** in this case a branch in the program will be missed.
- **Branch modification:** in this case the condition or destination of a branch in the program is changed.
- **Branch insertion:** in this case the program jumps to an illegal point.
- **Program counter manipulation:** in this case the program counter of a program is changed randomly and leads to random jumps.

The method of this paper can detect control flow and data errors simultaneously and force less overhead on memory and execution time of the program. The previous techniques have not considered these parameters with each other and usually focus on fault coverage rather than memory and performance overheads. Time/space overhead is as important as fault coverage and should be considered in different methods precisely. The contributions of this paper can be listed as follows:

- Presenting a new control flow checking method which has better fault coverage, memory and performance overhead than other techniques.
- Presenting a new data error detection method that limits the overhead of replication and repeats more critical variables instead of the whole program. In this way, the tradeoff between memory, performance overhead and fault coverage will be satisfied better than full duplication methods.
- Combining the former methods with each other to construct a soft error tolerant system that detects about 96.3% of the errors and forces less than 80% memory and performance overhead to the system.
- Introducing a new parameter which is called "*evaluation factor*" and considers fault coverage, memory and performance overhead simultaneously.

In the next section, the proposed method of this paper will be explained.

3. The Proposed Technique

3.1. Description of Control Flow Error Detection Method

In the previous sections, the basic idea of control flow checking methods is explained. Because of advantages of software-based techniques, the method of this paper is in this category. Like other control flow checking methods, this technique divides the program into some basic blocks and assigns a signature to each block based on its location in control flow graph. A basic block is consisted of maximum number of instructions which are run continuously. Therefore, the instructions of basic blocks are not branches or destination of branch instructions. In this way the program *P* can be shown as a graph which

contains some nodes and edge sand is called control flow graph. Each node represents a basic block and the edges show the transitions between blocks. **Figure 1(a)** shows a sample program and **Figure 1(b)** is its control flow graph.

In the control flow graph of **Figure 1(b)**, the successors of node 1 are 2 and 3 so node I is the successor of node J if there is a legal branch from node I to J.

In the proposed technique of this paper like other software-based methods, for control flow checking, a signature is assigned to each basic block. In SEDSR, this signature is variable S_i that shows successor blocks of the present block. For control flow error detection in basic blocks, four redundant instructions are also defined. The first one is called *control* and inserts at the beginning of each basic block to verify the correctness of branch destination.

The second one is called *check* and confirms the correctness of destination it also checks if the present block is one of the successors of the source basic block. For checking the correctness of the accessibility, Equation (1) is utilized:

$$error = S[sel]; \quad (1)$$

S is S_i variable that is updated during the program execution and sel shows the number of current basic block. At the middle of each basic block, if sel^{th} bit of S which shows the present basic block number equals 1, the destination is assigned correctly. Otherwise, *error signal* that shows the transient fault occurrence is activated and the program will be halted.

The third instruction is called *update* and updates S at runtime. For updating control flow signature, Equation (2) is designed which is run at the middle of each basic block:

$$S = s_i \quad (2)$$

Therefore, variable S is updated at the middle of basic blocks and is prepared to go to the next destination. It should be noted that S is set to 00000…1 the first time to go to the first basic block and other jumps become impermissible for it.

The last instruction is called *exit* and is run at the end of each basic block. It updates the amount of *sel* variable to the number that shows the current basic block.

Update instruction is placed at the middle of each basic block and some of the errors caused by illegal intra jumps in a specific basic block are detected. Putting the redundant instructions at the middle of basic blocks is similar to break each block into two parts. After error detection, *error signal* equals 0 and the program will be stopped.

If the illegal jump is occurred inter a basic block, this error will be detected by the *check* and *control* instructions in the successor blocks. **Figure 2** shows the first basic block of sample program of **Figure 1**, after adding redundant instructions.

This method is much simpler than the previous techniques because it uses no logical operations like AND or XOR for control flow checking. Its ability in intra block control flow errors detection makes it more capable than other methods of this field. In the next section, the ability of this method is compared analytically and experimentally with the previous techniques.

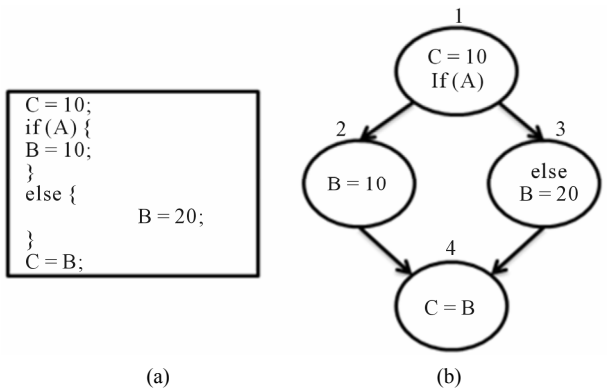

Figure 1. (a) A sample program; (b) Its control flow graph.

3.2. Description of Data Error Detection Method

A popular method for soft error detection in programs is data and instructions replication. These kinds of methods are generally expensive in terms of memory size, execution slow down and programming limitations. However, they have enormous capability in fault detection. In many applications like real time systems, the memory storage size and the execution time are limited and if these constraints don't meet, the system's performance degrades a lot and in many cases its output is not desired.

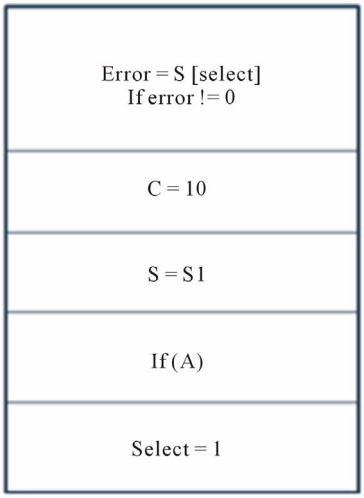

Figure 2. Sample basic block after adding redundant instructions.

As it was explained in the previous sections, control flow graph of each program shows its basic blocks and the dependency between different parts of it. The method that is proposed in this paper for data error detection determines a critical block in the control flow graph that has the most number of fan outs. This block is critical because its results propagate to other parts of the program and can affect other blocks in a faulty way. So isolating this part can decrease the faults of the program.

It is mentioned that full program duplication methods are not appropriate because of their overheads in execution time and memory. These days limiting the duplication scope is a good replacement for full duplication methods in many classes of applications such as real time and general purpose ones.

According to principle of locality in software, it is proved that 90% of errors are embedded in 10% of the program code [12]. Determining this critical section of code that contains most of the faults is very complex and dependent to the application. But by detecting this critical part, and duplicating it, the appropriate results in fault coverage, performance and memory overhead will be achieved simultaneously. There are many criteria for critical region determination. In this paper, the critical block of every program based on the number of fan outs have been selected. Our evaluations show that the number of fan outs is a suitable parameter in lots of benchmark sand has an acceptable result in fault coverage, performance and memory overhead.

In SEDSR, a block that has the most fan outs is determined and duplicates in compile time. At this time, the variables of the critical block are divided into two categories: middle and final variables. The middle ones are important in computing the other variables but final variables are not participating in any computations. In the critical block, a redundant instruction is inserted after final variables to compare these parameters in the original and replicated blocks. If there is any mismatch between these variables in two blocks, an error is reported and the program execution will be stopped.

Critical block has the most connections with other parts of the program and so an error in its output propagates and infects other blocks. This block duplication can detect a great percentage of errors and forces less overhead in memory and execution time to the program.

Figure 3 shows a sample control flow graph. As shown in this graph, block A is critical because it has the most fan outs in the graph and its results propagate to other parts and can defect them. By duplicating this block and executing the redundant along with the original version, faults are detected and the execution time and memory space remain at an acceptable amount.

The data error detection of this paper is compared with a full duplication method that is presented in [6]. Full duplication method is prevalent because of its high fault coverage and is used in RSCFCDV method too [8]. In this technique, the full program is duplicated and the comparison instruction is inserted after final variables as it was explained before.

Figure 4 shows the process of this method in a sample program. In this program, a, b, c are the middle variables and d is the final one. In this way, the comparison instruction is placed after writing on final variable and a great percentage of the occurred errors on data are detectable. In this way overheads because of comparison instruction will reduce in comparison to other methods like [4,5]. So this method has the best overhead between all of the full duplication methods and SEDSR will compare with this technique in the next section.

4. Evaluation Results

In this section, the evaluation results of proposed methods for control flow and data error are presented based on the analysis and experiment. SEDSR is compared with RSCFCDV method that has two parts for control flow and data errors.

4.1. Analytical Evaluation

For analytical evaluation of control flow checking method, all possible jumps are considered and the capability

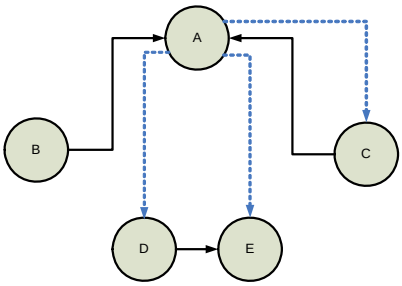

Figure 3. Control flow graph of a sample program.

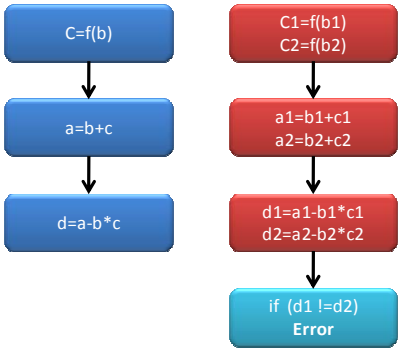

Figure 4. (a) Usual execution of a sample program; (b) Control flow graph of sample program after adding comparison and redundant instructions.

of SEDSR and RSCFCDV for error detection is compared with each other. There are nine different cases that are shown in **Table 1**. seven cases out of nine are illegal jumps that occur between basic blocks and the other ones are in a specified block. In **Table 1**, B_i and B_j are basic blocks of a sample program.

As it can be derived from this table, the control flow checking method that is proposed in this paper is much more capable in error detection than RSCFC [7]. The new method inserts a control instruction at the beginning of each basic block and in this way detects all illegal jumps to the beginning of the blocks. But in RSCFC, the detection of illegal jumps to the beginning of a basic block requires SET instruction execution. On the other hand, the proposed control flow checking can detect some percentages of intra block illegal jumps. So from all nine cases which are shown in **Table 1**, the proposed technique of this paper can do right at seven of them and RSCFCDV at three of them. This analytical evaluation of two methods shows that the proposed method of this paper does much better for control flow checking because of its design and redundant instructions which are placed in efficient places of basic blocks.

4.2. Experimental Evaluation

For experimental evaluation of the proposed method, fault injection is used. **Figure 5** shows the fault injection environment that contains the following elements:

- A Background Debug Mode module that can be utilized for both programming and debugging. It can also be used for fault injection like [13].
- Development board phyCORE-MPC555 [14].
- A personal computer.

The fault injection methods are divided into three main categories as follows:

- Direct fault injection on to processor registers by using BDM module.
- Deletion of existing jumps instruction of the program.
- Applying redundant jump instructions to the program.
- Changing jump instructions of the program or their operands.

According to the effects of injected faults in the program, five different cases are produced:

- **CR (Correct Result):** the fault doesn't change the final result of the program.
- **OS (Operating System):** the fault is detected by operating systems and its exceptions.
- **WR (Wrong Result):** the fault change the final result of the program and produces a wrong output.
- **TO (Time Out):** the fault change program execution time and it does not end in a specified amount of time.
- **SD (Single Detection):** the fault is detected by the instructions that are used for control flow checking.

The fault coverage of every method is equal to its SD percentage and the other kinds of detections like TO and OS is not the part of technique's detection capability.

Fault injection operation is applied to seven benchmark programs to compare the proposed method of this paper with RSCFCDV [8] which considers control flow and data errors simultaneously. These benchmarks are Bubble Sort (BS), Quick Sort (QS), Matrix Multiplication (MM) and Linked List Insertion (LLI) which are typical benchmarks that are used in previous researches of this field [1,7,9] and gzip, parser and earthquake that are standardized integers and floating pointSPEC2000 programs [14].

Three main parameters are computed based on fault injection results which are fault coverage, memory and performance overheads. Fault coverage is defined as the percentage of detected faults that do not damage the final outputs of the program. Performance overhead represents the execution time of fault tolerant version of the program to this parameter in original one. On the other hand, memory overhead is defined as memory capacity of the program with redundant instructions to this parameter at original code.

For experimental evaluation 1000 faults are injected to each of the mentioned benchmarks and the detection capabilities of methods are compared to each other. Every fault that is injected to the program will have one of the following effects on the final output of the program: 1) **CR:** the fault that is injected to the program do

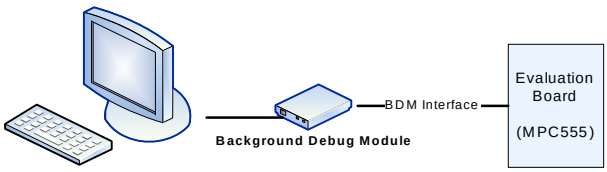

Figure 5. Fault injection mechanism structure by the use of BDM.

Table 1. Detection capabilities comparison between SEDSR method and RSCFC (BEG: begin; MID: middle).

	Beg-Beg $B_i \rightarrow B_j$	Beg-Mid $B_i \rightarrow B_j$	Beg-End $B_i \rightarrow B_j$	Mid-Beg $B_i \rightarrow B_j$	Mid-Mid $B_i \rightarrow B_j$	Mid-End $B_i \rightarrow B_j$	End-Beg $B_i \rightarrow B_j$	End-Mid $B_i \rightarrow B_j$	End-End $B_i \rightarrow B_j$	Beg-End $B_i \rightarrow B_i$	End-Beg $B_i \rightarrow B_i$
RSCFC	✗	✗	✗	✓	✓	✗	✗	✓	✗	✗	✗
SEDSR	✓	✗	✓	✓	✗	✓	✓	✓	✓	✓	✓

not change the final result of the program; 2) **OS:** the fault is detected by an operating system exception; 3) **WR:** the fault makes final output of the program incorrect; 4) **TO:** the fault creates an infinitive loop in the program and the program times out; and 5) **SD:** the fault is detected by redundant instructions that are added to the program.

Table 2 shows the results of this fault injection and **Table 3** represents the memory and performance overhead of different methods.

As it can be derived from the tables, the overall memory and performance overhead of the proposed method of this paper is less than these parameters in RSCFCDV. Memory and performance overheads are very critical in systems that have limited memory and their execution time is very critical because of their real time tasks. Fault coverage of the proposed technique is very close to this parameter in RSCFCDV because of the capability of control flow checking method of this paper that operates better than RSCFC and the data error detection technique detects the great percentage of faults with very less overhead.

In this part we will define a new parameter that is called "Evaluation Factor" which considers fault coverage, memory and performance overheads simultaneously. As we know the best method has high fault coverage and low overheads. The evaluation factor is defined as follows:

Table 2. Fault injection results of RSCFCDV and SEDSR methods (CR: correct results; OS: operating system; WR: wrong results; TO: time out; SD: single detection).

	CR	OS	WR	TO	SD
RSCFCDV-BS	48.12	24.87	0.61	0.52	25.88
RSCFCDV-QS	43.46	28.76	0.45	0.41	26.92
RSCFCDV-MM	45.78	26.12	0.76	0.66	26.68
RSCFCDV-LLI	46.82	23.07	0.34	0.76	29.01
RSCFCDV-gzip	39.16	29.78	1.89	029	28.88
RSCFCDV-parser	40.51	28.34	0.67	0.35	30.13
RSCFCDV-earth	36.48	30.93	1.06	0.41	31.12
SMSD-BS	42.34	25.12	4.56	0.43	27.55
SMSD-QS	49.65	19.76	3.76	0.65	26.18
SMSD-MM	44.28	20.45	5.12	0.93	29.22
SMSD-LLI	43.32	25.09	4.08	0.65	26.86
SMSD-gzip	39.03	28.34	3.16	0.92	28.55
SMSD-parser	48.12	21.54	2.48	0.16	27.7
SMSD-earth	44.05	22.89	3.03	0.39	29.64

Table 3. Memory and performance overhead of SEDSR and RSCFCDV methods.

Program	Memory Overhead		Performance Overhead	
	RSCFCDV	SMSD	RSCFCDV	SMSD
BS	1.83	1.27	1.67	1.12
QS	2.02	1.31	1.84	1.04
MM	1.98	1.46	1.71	1.21
LLI	2.01	1.16	2.12	1.02
gzip	2.11	1.45	1.98	1.32
parser	2.21	1.32	1.91	1.09
Earthquake	2.14	**SMSD**	**RSCFCDV**	**SMSD**

$$\text{Evaluation Factor} = \frac{\text{Fault coverage}}{\text{Memory overhead} \times \text{Performance overhead}}$$

The method which has greater Evaluation Factor is better because it considers the tradeoff between different parameters. A good soft error tolerance method should have a high fault coverage and low space and time overheads. **Figure 6** shows the Evaluation Factor of two mentioned methods. As it can be derived from this figure, the method that is proposed in this paper has greater Evaluation Factor than RSCFCDV and is more appropriate for using in safety-critical applications.

5. Conclusion

This paper presents a new software-based method for soft error detection. Transient faults cause control flow and data errors in systems that violate the final outputs of a system. The method of this paper considers both control flow and data errors by dividing the program to some basic blocks and duplicating execution in a specified block. A new metric that is called "evaluation factor" is introduced in this paper that considers fault coverage, memory and performance overheads simultaneously. Experimental results of fault injection show that the proposed method of this paper has better in evaluation parameter than other methods of this field. The main novelty of this paper is presenting a software based soft error detection method that is able to detect control flow and data errors. This technique has much less memory and performance overheads than other methods of this field and is an appropriate choice for real time safety critical applications which have limitations in memory and execution time.

6. Acknowledgements

This paper was supported by the Research Institute for

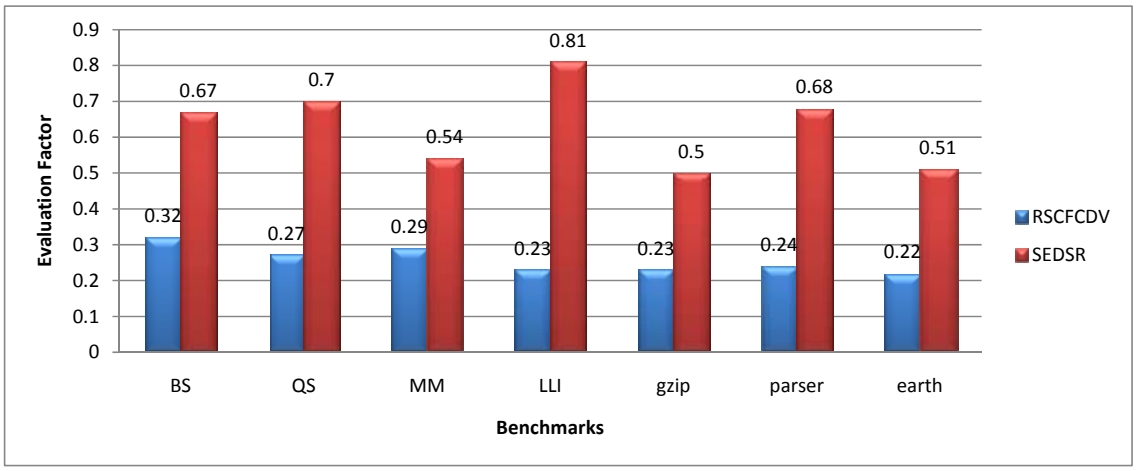

Figure 6. Evaluation factor of RSCFCDV and SEDSR.

ICT-ITRC.

REFERENCES

[1] N. Oh, P. P. Shirvani and E. J. McClusky, "Control Flow Checking by Software Signature," *IEEE Transaction on Reliability*, Vol. 51, No. 2, 2002, pp. 111-122. doi:10.1109/24.994926

[2] A. Mahmood, "Concurrent Error Detection Using Watchdog Processors—A Survey," *IEEE Transaction on Computers*, Vol. 37, No. 2, 1988, pp. 160-174. doi:10.1109/12.2145

[3] A. Rajabzadeh and G. Miremadi, "CFCET: A Hardware Based Control Flow Checking Technique in COTS Processors Using Execution Training," *Elsevier Journal on Computer Microelectronics and Reliability*, Vol. 46, 2006, pp. 959-972.

[4] N. Oh, P. P. Shirvani and E. J. McCluskey, "Error Detection by Duplicated Instructions in Super-Scalar Processors," *IEEE Transaction on Reliability*, Vol. 51, No. 1, 2002, pp. 63-75. doi:10.1109/24.994913

[5] N. Oh, M. Subhasish and E. J. McCluskey, "ED4I: Error Detection by Diverse Data and Duplicated Instructions," *IEEE Transaction on Computers*, Vol. 51, No. 2, 2002, pp. 180-199. doi:10.1109/12.980007

[6] B. Nicolescu and R. Velazco, "Detecting Soft Errors by a Purely Software Approach: Method, Tools and Experimental Results," *Proceedings of the Design, Automation and Test in Europe Conference and Exhibition*, Munich, 3-7 March 2003.

[7] A. Li and B. Hong, "On-Line Control Flow Error Detection Using Relationship Signatures among Basic Blocks," *Elsevier Computers and Electrical Engineering*, Vol. 36, No. 1, 2010, pp. 132-141. doi:10.1016/j.compeleceng.2008.08.010

[8] A. Li and B. Hong, "Software Implemented Transient Fault Detection in Space Environments," *Elsevier Aerospace science and technology*, Vol. 11, No. 2-3, 2007, pp. 245-252.

[9] 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," *IEEE Transactions on Parallel Distributed Systems*, Vol. 10, No. 6, 1999, pp. 627-641. doi:10.1109/71.774911

[10] J. L. Li, Q. P. Tan and J. J. Xu, "A Software-Implemented Configurable Control Flow Checking Method," *International Symposium on Parallel Archirectures, Algorithmes and Programming*, IEEE Computer Society, Washington, 2010, pp. 199-205.

[11] A. Vemu and J. Abraham, "CEDA: Control-Flow Error Detection through Assertions," *IEEE Transaction on Compuer*, Vol. 60, No. 9, 2011, pp. 1233-1245. doi:10.1109/TC.2011.101

[12] A. S. Tanenbaum, J. N. Herder and H. Bos, "Can We Make Operating System Reliable and Secure?" *IEEE Magazine*, Vol. 39, No. 5, 2006, pp. 44-51.

[13] S. A. Asghari, H. Pedram, H. Taheri and M Khademi, "A New Background Debug Mode Based Technique for Fault Injection in Embedded Systems," *International Review on Modeling and Simulation*, Vol. 3, No. 3, 2010, pp. 415-422.

[14] J. Henning, "SPEC CPU2000: Measuring CPU Performance in the new Millennium," *Computer*, Vol. 33, No. 7, 2000, pp. 28-35. doi:10.1109/2.869367

