An Effective Soft Error Detection Mechanism using Redundant Instructions

Seyyed Amir Asghari[1] and Hassan Taheri[2]
[1]Computer and Electrical Engineering Department, Kharazmi University, Iran
[2]Electrical Engineering Department, Amirkabir University of Technology, Iran

Abstract: *Computer Systems which operate in space environment are Subject to different radiation phenomena that lead to soft errors and can cause unpredictable behaviours of computer-based systems. Commercial Off-The Shelf (COTS) equipment which is commonly used in space missions cannot tolerate some threats such as Single Event Upsets (SEU). Therefore, there are some considerations in resisting this equipment against possible threats. In this paper, a software instruction level method that is called Soft Error Detection using Redundant Instructions (SEDRI) is provided to detect soft errors which influence control flow and program data. This method is evaluated by fault injection on several C benchmark programs. The experimental results show that without protecting a program against control flow and data errors 34% of them affect the program and damage it; but, by using our method, this rate is decreased to about 11%. Comparing to previous presented techniques, SEDRI method has a considerable improvement in performance and memory overhead, i.e., 46% and 55% respectively, and its fault coverage decrease about 9%.*

Keywords: *Control Flow Checking, data error, fault coverage, soft error, software-based error detection.*

Received October 28, 2012; accepted January 18, 2013, published online April 17, 2014

1. Introduction

Scaling of Very Large Scale Integration (VLSI) technologies, coupled with increased integrated circuit complexity, will strongly increase the occurrence of transient faults (also known as soft errors) [9]. Particularly, this is true with respect to computers operating in space environment, which are subject to different radiation phenomena. Transient faults, unlike manufacturing or design faults do not occur consistently. Instead, these faults are caused by external events, such as electromagnetic interferences, power glitches, or highly energized particles striking the chip. These events do not cause permanent physical damage to the chip, but can alter signal transfers or stored data and thus cause incorrect program execution [15]. Single Event Upsets (SEUs) is one of the most common reasons of transient faults in space missions that use Commercial Off-The Shelf (COTS) equipment.

Transient faults have caused lots of significant failures for example FengYun-1(B) meteorology satellite launched in China in 1990 went out of commission ahead of schedule because of attitude control system's losing control caused by SEUs [6, 15]. In 2000, Sun Microsystems acknowledged that cosmic rays interfered with cache memories and caused crashes in server systems at major customer sites, including America Online, eBay, and dozens of others [6].

Generally, hardware or software redundancy can be introduced to handle transient faults. But, the hardware-based methods are expensive, since they require replicated hardware modules or developing custom hardware equipped with error detection mechanisms that verify operation correctness on line. When development cost is a major concern, as in low volume applications, designers tend to adopt commercially available hardware, even in the case of safety-critical applications. In this context, software based fault tolerance is an attractive solution, since it allows implementing dependable systems without incurring the high costs associated with developing custom hardware-based tolerance techniques not readily available in COTS products [9].

Transient faults induced by hardware have an impact on software running on it, which causes either control flow or data errors in software. Control Flow Errors (CFEs) change the control flow of the application and can be defined to be any fault that causes a divergence from the sequence of program counter values seen during the fault-free execution of the application. For these kinds of faults, software techniques proposed are mainly based on signature analysis, where a unique signature is associated to a basic block at pre-compile time. During program execution, the same signature is computed and compared with a reference signature [1, 5, 7, 8, 9, 11,12, 13].

On the other side, data errors affect the values of data variables, registers, or memory locations used by the application [10]. To address this kind of faults, the proposed approaches rely on information redundancy to store multiple copies of the same information and on the introduction of consistency checks whose purpose is to verify the coherence of the replicated information. Some presented solutions in this field can be found in served references [6, 10, 13, 12, 14]. The method that is presented in this paper has low memory and performance overhead that is very important for some applications [2].

Most of previous approaches proposed so far cannot give us a satisfying result about fault detection rate and time/space overhead. Some of the above approaches either have high detection rate, but with great overhead (e.g, RSCFCDV method in [6]). Nicolescu *et al.,* [9] claim that their method can provide full coverage against SEUs, but the program execution speed decreases by a threefold factor and required memory is four times larger, which cannot be useful in our target applications.

In [6] a control flow checking method is combined with data duplication and detects about 99% of faults that damage control flow and data of the program. But because of duplicating the whole program, its memory and performance overhead rate is about 1.8 and 1.5 that are not applicable in the mentioned applications.

This paper proposes a new solution that not only has an acceptable fault coverage for control flow and data error detection, but also the mean increases of both the execution time and occupied memory are less than other methods. The novelty of the proposed method consists in the adopted control flow checking technique, called Soft Error Detection using Redundant Instructions (SEDRI), which is based on the partition of programs into basic blocks. Firstly, it assigns a signature to each basic block, into which the successors of every block are encoded. Control flow faults are detected through comparing the run-time signature of current block with the expected value with extra instructions induced in the middle and the end of each block. We assessed the fault detection capabilities of our approach on 4 simple C benchmark programs. The results show that the hardened program is able to detect most of the injected faults with appropriate time and space overhead.

In the second section of this paper, the previous works that try to detect soft errors are explained and studied. The third section of the paper introduces the proposed method and its detection capabilities. The experimental results of the proposed method are explained in the fourth section.

2. Literature Review

Transient fault occurrence in computer systems and during their running leads to considerable damages. For achieving reliability in computer systems, CFEs detection is one of the effective techniques. For detecting such errors, many techniques have been presented since 1980 that can be divided into two categories of hardware and software based techniques.

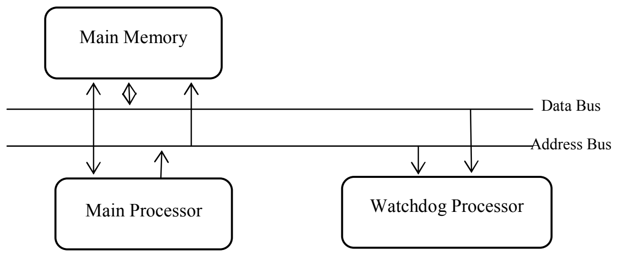


Figure 1. The structure of a system with watchdog processor.

In this category of methods, the control flow of a program is checked by assigning a signature to each basic block and sending the signatures of the beginning and the end of basic blocks to the watchdog processor [11]. The first research in this field carried out in 1990. One of the primary methods for detecting transient faults in processors are using watchdog processors. Watchdog processor is a kind of co-processor that detects system faults by monitoring the main processor behavior. This idea is actually a generalization of watchdog timer and is implemented in the system level. Figure1 shows the general structure of a system that uses watchdog processor [7].

In the presented software-based methods, the general procedure of the operation is similar to the previous methods and based on the application and there is possibility of detecting three kinds of control flow checking errors. Software-based methods are usually performed using on data or code replication and can be placed at the procedure or statement level [1, 5, 7, 10, 13]. In these methods, program is divided into some basic blocks and for each section, a signature is assigned.

Basic block is a maximal set of ordered instructions that run serially and do not contain any jump, call or branch instruction except for the last one, also instructions of a basic block should not be the destination of any jump or branch except for the first one. By dividing the program to basic blocks, it can be represented as a graph that composes the set of V nodes and i edges where $V=\{v_1, v_2... v_n\}$ and $E=\{e_1, e_2... e_m\}$. Each node of the program graph, shows a basic block and each edge e_x shows a branch from node v_i to the node v_j. As an example we show the control flow graph of a sample program in Figure 2. In Figure 2, the instruction number 2 is a conditional command and so is a divider between basic blocks. The instruction number 5 that is the destination of a conditional jump is used for partitioning between blocks. In this way and as it can be seen in Figure 2, the main program is divided into three basic blocks [4].

In some special applications like space systems, due to beam and heavy ion radiation, many soft errors lead to disorder in control flow of a software systems and therefore unpredictable behaviour. Many studies have been done on the effect of protons and heavy ions (atoms that are heavier than helium) on electronic circuits. One of the main effects of these protons and ions is SEU. Due to the special space environment and the urgent need of equipment of this environment to a high reliability, many works have been done for enhancing the reliability of space equipments [4].

A typical CFEs detection technique maintains one or more run time signature registers. Let *S* be the set of run time registers. *S* is continuously updated and verified as the program is running and any inter-node CFEs is detected. At compile time, the program is first prepared and its program graph is built as shown in Figure 2.

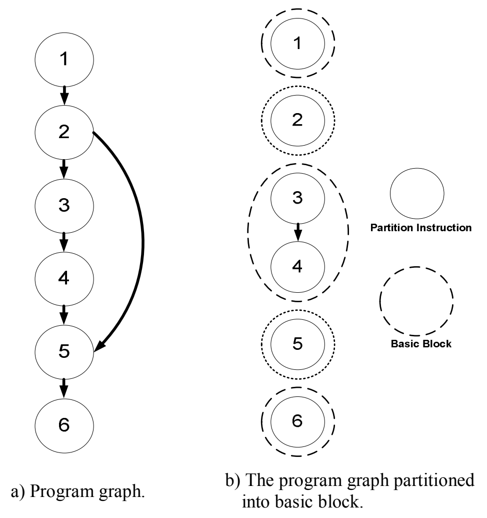


Figure 2. Program division into some basic blocks [3].

Then, signatures are assigned to each point (inside the nodes and along the edges) in the program graph. These signatures are values that *S* is expected to have at run time when execution reaches corresponding points in the program. At run time, *S* is legal if it equals to its expected value and illegal when some extra instructions are inserted in the program for updating *S* at the beginning and end of each basic block. These instructions continuously check the legality of *S* and update it. *S* is assigned to its expected value at the start of the program execution. The nature of the update and check instructions implemented for a particular technique determines the performance and memory overheads associated with it.

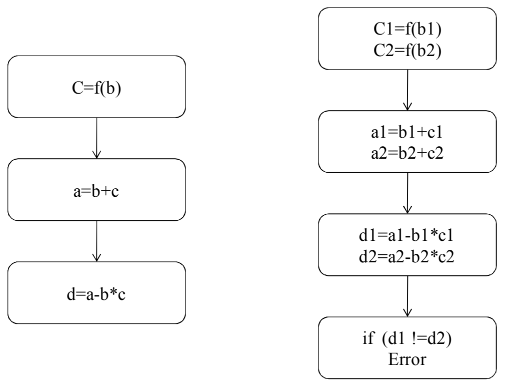
One of the recent works in this field is RSCFC [5] in which the relationship between blocks is extracted and then based on the kind of the relationship, a signature is assigned to each block in which the existed relationships are coded. The faults in the flow control program are detected by ANDing runtime signatures with the information at the beginning and end of the blocks. In CFCSS [12] method, a global variable of *G* is also added to the program which is consisted of the running block signature amount. During the program, whenever it enters to a new basic block, *G* is updated to a new amount. ECCA [1] technique, that is another CFC method adds a prime number to each basic block and checks control flow graph of a program.

For detecting data errors that consist 30% of soft errors, many methods have been delivered that are generally based on information redundancy and repeating the running program variables and the main difference between them is the place of comparison instructions. One of the most prominent methods for detecting data errors is EDDI [13]. Instruction redundancy in the presented technique has no influence on the program output, but it detects errors during program running. The main idea is repeating the instructions in registers and several variables. The values of these two registers are compared with each other and the error is reported in case of incoherence (mismatch of two variables). Some comparison instructions check the coherence between the main running and redundant that the situation of these instructions is very important. This comparison is performed exactly before store instructions or jump in the memory. Time scheduling between main instructions and the redundant in this method is according to list scheduling algorithm [13]. ED^{4}I method [14] is a Software Implemented Fault Tolerance (SIFT) method that can detect transient and temporary errors by running two different programs but by different data collections that implement one functionality and comparing their output. Transient errors that cause malfunctions in one of the programs can be detected by this method. Some of these transient errors are the transient errors in processors and bit-flips that occur in memories. Bit-flip is an unwanted change in memory cells and is created based on several elements that SEUs are one of these elements. For example, bit-flips that happen in program code section during program running can change program behavior and lead to inaccurate results. By comparing inaccurate results (caused by erroneous program) and accurate results (caused by not erroneous programs), an error (in this case, bit-flip) can be detected. When both main and redundant programs produce inaccurate output, ED^{4}I technique can detect the error by continuing program running till the program outputs differ with each other. The point that needs to be considered here is that this technique is not able to detect the errors that lead the program to place in an unlimited loop and cannot exit it (this case usually happens when the registers like Instruction Pointer (IP) confront with undesirable changes and the next program IP jumps to an inaccurate place). These errors are usually called CFE Program (the programs that disorder the accurate program running). For detecting these errors, software control flow checking techniques or watchdog timers are utilized.

Another technique that is presented in [6] like the method of this paper considers data and CFEs simultaneously. It utilizes RSCFC for detecting CFEs and detects data errors by using instruction repetition and their results comparison. For detecting such errors, the relationship between variables are extracted and divided into two groups:

- Middle variables that are important for computing other parameters.
- Final variables that do not participate in computing any other parameter.

After categorizing the variables into two mentioned groups, all of the variables are repeated and after each writing action in final variables, an instruction should be placed for comparing main and repeated variables. In case of any mismatch between these two values, a data error is detected. Figure 3 shows this method in a sample program. In this program a, b, and c are middle variables and d is a final one. At the end of computation, if the final variables are not equal, an error is reported. Therefore, comparison instruction is only placed after one final variable during writing. In this way, a great percentage of occurred errors on program data are detectable and the overhead of the redundant instructions is less than the similar mentioned methods.



a) A simple program. b) Data replicated program.

Figure 3. A Sample of software redundancy to data error detection.

The presented method in experimental analysis has 99% fault coverage with performance and memory overhead rate of about 1.5 and 1.8. The value of memory and performance overhead is high because of full duplication of variables and instructions.

3. The Proposed Technique

In this section two techniques for CFEs and data errors detection are presented.

3.1. The Proposed Technique for Control Flow Error Detection

In the previous section, some of the techniques for CFEs detection based on software and hardware redundancy were analyzed. Control flow checking technique of this paper is delivered in the following. With regard to the mentioned advantages of software-based methods, the presented technique in this paper is based on software and like the previous techniques; it divides the program code to some basic blocks and specifies a signature for each block.

In the proposed technique of this paper like most methods, for detecting CFEs, a signature is assigned to each basic block. This signature is variable S_i that shows successor blocks of the current block. For control flow checking in basic blocks, four instructions are defined and a unique signature is assigned. For a program flow graph $P=\{V, E\}$, we define $Suc(vi)$ as the set of nodes successor of v_i if and only if $bri, j \in E$, then the node $vj \in Suc(vi)$ [3]. The signature s_i that is assigned to basic block vi is some kind of its successor set representation. A sample of assigning signature to basic blocks of the program is shown in the following:

$$Suc(vi)=\{Vj, Vk, Vm, Vn\} \text{ therefore}$$
$$Si=1(\text{for } n)0\ldots01(\text{for } m)\ 0\ldots01(\text{for } k\ \ 0\ldots1(\text{for } j)\ 0.$$

As it can be derived from this sample, the signature of each basic block has N bit where N is the number of basic blocks of the program. The n^{th} bit of signature stands for basic block n and it is equal to 1 if basic block n is one of the successors of block vi. In this way the transformation information between basic blocks, encode in the signature and can be used for control flow checking. Figure 4 shows a sample control flow graph and the signature of each node according to the procedure that is described.

Four instructions are added to each basic block for the purpose of control flow checking. The first instruction is called control and is placed at the beginning of each basic block. The role of this instruction is entry verification for each basic block. If an unwanted jump transfers the control of program to the beginning of an illegal block, then control instruction can detect error and stop the program.

The second one is called check and its duty is to confirm that the destination is assigned correctly and the present block is one of the successors of the source block.

For checking the correctness of the accessibility, Equation 1 is utilized:

$$err = S\,[Sel] \tag{1}$$

S is S_i variable that is updated at run time. At the beginning of each basic block, if Sel^{th} bit (that shows the present basic block number) equals 1, the destination has been assigned correctly. Otherwise, *err Signal* that is a sign of error is activated. Another instruction is called *update* and its purpose is to update the S variable. For updating run time signature, Equation 2 is utilized:

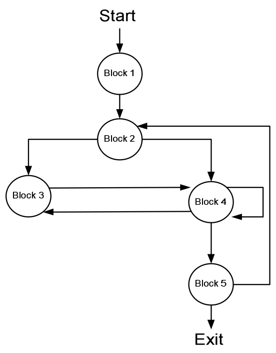
$$S = S_i \tag{2}$$

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

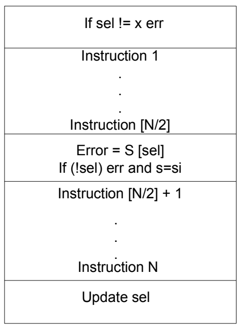
The last instruction that is added to each basic block is called *exit* and is run at the end of each basic block and updates the amount of *Sel* variable to the number that is the sign of the present basic block. Variable *Sel* is an integer value which is updated at the end of each basic block and is used in the next redundant instructions.

The check and update instructions are placed at the middle of each basic block and in this way some of the errors caused by impermissible interior jumps in a basic block are detected. By placing these instructions at the middle of each basic block, some percentage of illegal jumps from a specified basic block to itself is detected too; this case will be described in this paper. After detecting error, *err Signal* equals 0 and the program will be stopped. Control flow graph of a sample program is shown in Figure 4-a. In this figure the signature of each basic block is also specified based on its successors. In this graph, basic block interconnections and their interactions are also specified. The structure of a basic block and its redundant instructions is shown in Figure 4.b. In this figure, "*Sel*" is shown the block number and x represents the current block identifier.

By inserting redundant instructions, most of the single errors due to incorrect jumps can be found. The proof of this claim is shown later in this section.



a) Control graph of a sample program.



b) The interior structure of a basic block by inserting redundant instructions.

Figure 4. The mechanism of control flow checking.

3.2. The Proposed Technique for Detecting Data Errors

The goal of this mechanism is detecting the errors that influence data and values of the program. As mentioned in the previous section, a popular method for Soft Error Detection of programs is data and instructions duplication. In section 2, full duplication methods and their improvements in fault coverage have been reviewed. However, due to their memory and speed overheads, it seems that full data replication is not a good way and a limited replication scope is more useful in general purpose applications that cost is important. On the other hand, according to the principle of locality in software, it is proved that 90% of errors are embedded in 10% of the code [16]. Detecting the critical section of software is so complex and application-based. But, by detecting this critical part, and duplicating it, appropriate results in fault coverage, performance and memory overhead can be achieved.

In this section, an effective method is proposed for detecting critical section of the program. A block is determined and highlighted as critical because it has the most connections with other parts of the program and so an error in its output propagates and infects other blocks. This block has the most important part in program running and its duplication can detect a great percentage of errors and is called critical block.

In Critical Block Duplication (CBD) as mentioned, a critical block is determined from the control flow graph and is based on the number of fan outs of each block. A basic block that has the most number of fan outs is critical because its results propagates to many parts of the program and can affects other blocks in a faulty way.

Critical block replica is consisted of separate registers and variables and all of the instructions in it will run independently. At the end, the results of the original and replicated blocks are compared; in case of any mismatch, an error is reported and the program will stop.

Figure 5 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. So, in this way, the performance and memory overhead will improve and fault coverage will remain in an acceptable range for real time and general purpose applications.

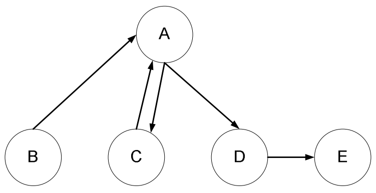


Figure 5. Control flow graph of a sample program.

3.3. Detection Capabilities of Control Flow Checking Technique

By inserting control, check and update instructions in every basic block, most of the single errors due to incorrect jumps can be found. The proofs of this claim are as follows:

1. One jump from *vi* node to *vj* node. When an impermissible and unwanted jump occurs from block *vi* to the block *vj*: In these case, when check instruction is run in node *vj*, since *Sel* variable amount in node *vj* is not updated, *err* variable is equal to 0 and this error is detected. For example, imagine in Figure 5-a, there is an unwanted jump from the end of the first basic block to the middle of the second one. In this case, since *Sel* variable at the end of the first block is not updated, it has the amount of zero. When the operation is reached to check and update instructions of the second basic block, *0th* bit from *S* variable that now has *signature_BB1* (equal to 0) is assigned to error signal. The error signal receives the amount of 0 and the error is detected. In this case: *S=Signature_BB1=00010,* S[*0*]*=0, err=*S[*0*]*=0.*
2. When an unwanted jump occurs from node *vi* to itself. When an unwanted jump occurs from an instruction before checking and updating to the instructions after them: In this case, since *S* variable has not been updated during running, error is detected in check instruction of the next block.

Table 1. Detection capabilities comparisons between our method and other techniques (beg: begin, mid: middle, Y: yes, and N: no).

	Beg-Beg (Inter Blocks)	Beg-Mid (Inter Blocks)	Beg-End (Inter Blocks)	Mid-Beg (Inter Blocks)	Mid-Mid (Inter Blocks)	Mid-End (Inter Blocks)	End-Beg (Inter Blocks)	End-Mid (Inter Blocks)	End-End (Inter Blocks)	Beg-End (Intra Blocks)	End-Beg (Intra Blocks)
CFCSS	Y	N	N	Y	N	N	Y	N	Y	N	N
ECCA	N	N	Y	N	N	Y	Y	Y	Y	N	N
RSCFC	N	N	N	Y	Y	N	N	Y	N	N	N
SEDRI	Y	N	Y	Y	N	Y	Y	Y	Y	Y	Y

For example, imagine an unwanted jump occurs from an instruction before check of the first block to after update instruction of the same block. In this state, update instruction has not been run so *S* variable takes its initial amount that is 00001. At the end of the first block, *Sel* variable is updated to 1 and the program enters the second basic block. In the second basic block and in its check instruction, the first bit of *S* variable is updated to 1 and it leads to detecting the occurred error: $S=S_{initial}=00001$, $err= S[Sel]=0$.

In Table 1, different cases of illegal jumps from a basic block to another and inside a block are considered. By inserting redundant instructions to the beginning, middle and end of a basic block, nine different cases will appear. Also, illegal jumps inside a specific basic block are shown in this table. All the cases are shown in Table 1 and detection capabilities of different methods of control flow checking field are compared with each other. As it can be concluded from this table, the detection capabilities of our method are better than others and it can handle most of cases. The two cases that are studied in this section are included in Table 1.

4. Experimental Evaluation

In this section, test development environment and experimental results have been explained.

4.1. Test Environment

For analyzing the control flow and data detection methods, the infrastructure shown in Figure 6 is utilized 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 [2].
- Development board phyCORE-MPC555.
- A personal computer.

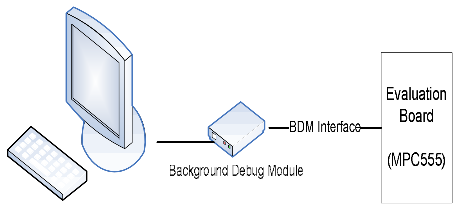


Figure 6. Fault injection mechanism structure by the use of BDM [2].

Different methods are used for fault injection. These methods are as follows:

- Direct fault injection onto processor registers by use of BDM module.
- Applying jump instructions (JMP, JL, JG, JNE, JLE, JGE, CALL and RET).
- Changing jump instructions.

Fault injection operation is applied for three benchmark programs of Bubble Sort (BS), Quick Sort (QS) and 40×40 Matrices Multiplication (MM) that 5000 faults are injected on them. Since, by the use of BDM, processor registers can be directly manipulated in the method of direct fault injection onto processor registers, it is considered to be a good solution with much higher speed and capability and is much closer to reality. For example, in this method, PC register can be directly manipulated. As it is shown in [4], by manipulating registers, exception occurrence probability is more than 76%. Therefore, this method does not provide an accurate test for error detection although it is much closer to reality. As a result, the second and third methods are more utilized for analyzing error detection method.

Moreover, for analyzing the proposed method, data segment, code segment bits changing are utilized randomly. For analyzing the proposed method, fault injection methods that explained are utilized. Table 2 shows the results of fault injection (Correct Result (CR), Operating System (OS), Wrong Result (WR), Time Out (TO) and Single Detection (SD)). RSCFCDV method has the problem of memory and performance overhead because as it was explained, in this method, two variables of $n+1$ defined, where n is the number of basic blocks and on the other hand, it inserts seven instructions to each basic block for CFEs detection. The volume of each basic block is about 3 to 5 instructions and inserting 5 redundant instructions and two $n+1$ bits variables to each basic block is not reasonable. On the other hand, it duplicates the whole program which is not applicable in many applications that cannot tolerate much overhead like.

In the proposed idea of this paper for control flow checking, four instructions are inserted in each basic block and an n bit variable is assigned as the signature as well as a one bit variable called *Sel* which is saved for operation. For data error detection, CBD method duplicates only one critical block and in this way it can detect a large amount of errors with very low memory and performance overhead.

By comparing size and speed of resisted programs with the normal ones, the results of Table 2 shows that proposed technique has performance and memory overhead of about 46% and 55% less than RSCFCDV method. On the other hand its fault coverage is 7% less than RSCFCDV because of the limited scope of duplication that is used in it. This amount of loss in fault coverage is acceptable in comparison with the gain of memory and performance overhead that are more important in embedded and real time applications. These kinds of systems have limitations in memory capacity and runtime that should be considered. The average fault coverage of traditional methods of Soft Error Detection is in about 96.43% as reported in [10, 11, 13].

Table 2. Results of fault injection in comparison with RSCFCDV method.

Benchmark	CR	OS	TO	WR	SD
BS	40.67	23.74	2.18	33.41	0
BS-RSCFCDV	42.56	31.87	1.75	1.09	22.73
BS-SEDRI	40.49	32.76	2.85	9.55	14.35
MM	37.86	25.12	2.36	34.66	0
MM-RSCFCDV	44.97	29.65	1.06	1.03	23.29
MM-SEDRI	42.48	30.09	1.29	8.87	17.27
QS	42.73	22.12	1.50	33.65	0
QS-RSCFCDV	43.87	34.21	2.03	0.97	18.92
QS-SEDRI	42.98	30.76	2.98	7.48	15.8

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

- *CR*: The fault doesn't change the final result of the program.
- *OS*: the fault is detected by operating systems and its exceptions.
- *WR*: The fault changes the final result of the program and produces a wrong output.
- *TO*: The fault change program execution time and it does not end in a specified amount of time.
- *SD*: 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.

Table 3 shows the performance and memory overhead of the proposed method. As it can be seen, this technique is much better than RSCFCDV in these parameters.

Table 3. The comparison between memory and performance overhead of SEDRI method and RSCFCDV method.

	Memory Overhead		Performance Overhead	
Program	RSCFCDV	SEDRI	RSCFCDV	SEDRI
BS	1.89	1.26	1.77	1.16
QS	1.91	1.38	1.23	1.09
MM	1.93	1.43	1.83	1.2
Average	1.91	1.36	1.61	1.15

5. Conclusions

Using COTS equipment is one of the appropriate choices in a wide range of applications such as space missions. However, without considering appropriate redundancy preparations in different levels (hardware, software, time, and information), this equipment cannot be utilized in space missions that are imposed to lots of dangers. In this paper, a new software implemented technique for Soft Error Detection is presented. This technique has two parts for control and data error detection. The main novelty of this method is in its low memory and performance overhead that is combined with an acceptable amount of fault coverage for these applications. In comparison with RSCFCDV technique that is presented in this field, SEDRI has 55% and 46% improvement in memory and performance overhead that is very important and valuable in the mentioned applications.

Acknowledgements

This paper was supported by the research institute for ICT-ITRC of Iran.

References

[1] Alkhalifa Z., Nair S., Krishnamurthy N., and Abraham A., "Design and Evaluation of System-Level Checks for On-line Control Flow Error Detection," *IEEE Transactions on Parallel Distributed Systems*, vol. 10, no. 6, pp. 627-641, 1999.

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

[3] Asghari A., Taheri H., Pedram H., and Kaynak O., "Software-based Control Flow Checking Against Transient Faults in Industrial Environments," *IEEE Transactions on Industrial Informatics*, vol. no. 99, pp. 1, 2013.

[4] Baumann C., "Soft Errors in Commercial Semiconductor Technology: Overview and Scaling Trends," *IEEE Reliability Physics Tutorial Notes, Reliability Fundamentals*, IEEE Press, USA, 2002.

[5] Li A. and Hong B., "On-line Control Flow Error Detection using Relationship Signatures Among Basic Blocks," *Elsevier journal of Computers and Electrical Engineering*, vol. 36, no.1, pp. 132-141, 2010.

[6] Li A. and Hong B., "Software Implemented Transient Fault Detection in Space Computer," *Elsevier journal of Aerospace Science and Technology*, vol. 11, no. 2, pp. 245-252, 2007.

[7] Mahmood A., "Concurrent Error Detection using Watchdog Processors-A Survey," *IEEE Transaction on Computers*, vol. 37, no. 2, pp. 160-174, 1988.

[8] Mammeri S. and Beghdad A., "On Handling Real-time Communications in MAC Protocol,"

the International Arab Journal of Infoormation Technology, vol. 9, no. 5, pp. 428-435, 2012.

[9] Nicolescu B., Savaria Y., and Velazco R., “Software Detection Mechanisms Providing Full Coverage Against Single Bit-flip Faults,” *IEEE Transactions on Nuclear science*, vol. 51, no. 6, pp. 3510-3518, 2004.

[10] Nicolescu B. and Velazco R., “Detecting Soft Errors by a Purely Software Approach: Method, Tools and Experimental Results,” *in Proceedings of the Design, Automation and Test in Europe Conference and Exhibition*, Munich, German, pp. 57-62, 2003.

[11] Nicolescu B., Savaria Y., and Velazco R., “Sied: Software Implemented Error Detection,” *in Proceedings of the 18th IEEE International Symposium on Defect and Fault Tolerance in VLSI Systems*, Boston, USA, pp. 589-596, 2003.

[12] Oh N., Shirvani P., and McClusky J., “Control Flow Checking by Software Signature,” *IEEE Transaction on Reliability*, vol. 51, no. 1, pp. 111-122, 2002.

[13] Oh N., Shirvani P., and McCluskey J., “Error Detection by Duplicated Instructions in Super-Scalar Processors,” *IEEE Transaction on Reliability*, vol. 51, no. 1, pp. 63-75, 2002.

[14] Oh N., Subhasish M., and McCluskey J., “ED4I: Error Detection by Diverse Data and Duplicated Instructions,” *IEEE Transaction on Computers*, vol. 51, no. 2, pp. 180-199, 2002.

[15] Reis A., Chang J., Vachharajani N., Rangan R., and August I., “Software Implemented Fault Tolerance,” *in Proceedings of the International Symposium on Code Generation and Optimization*, Washington, USA, pp. 243-254, 2005.

[16] Tanenbaum S., Herder N., and Bos H., “Can We Make Operating System Reliable and Secure?,” *IEEE Magazine*, vol. 39, no. 5, pp. 44-51, 2006.

Seyyed Amir Asghari He received his BSc degree in 2007 (hardware engineering major), MSc and PhD in 2009 and 2013 respectively (computer architecture major) from Amirkabir University of Technology. In 2013, He was a visiting researcher in Mechatronics Research Center, Bogazici University, Turkey. His current research interests include fault tolerant design, real time embedded system design, and operating systems.

Hassan Taheri received his BS degree from Amirkabir University of Technology in 1975, MS and PhD degrees in electrical engineering in 1975 and 1988 respectively from University of Manchester Institute of Science and Technology. He has served as a faculty member in the Electrical Engineering Department in Amirkabir University of Technology. He teaches courses in data communication network, computer communication, teletraffic engineering, electronic switching, digital communications, telephone switching, probability and statistics.