Research Journal of Applied Sciences, Engineering and Technology 4(21): 4367-4379, 2012
ISSN: 2040-7467
© Maxwell Scientific Organization, 2012
Submitted: April 17, 2012 Accepted: May 14, 2012 Published: November 01, 2012

I2BCFC: An Effective Intra-Inter Block Control Flow Checking Method Against Single Event Upsets

Seyyed Amir Asghari, Atena Abdi, Hassan Taheri, Hossein Pedram and Saadat Pourmozaffari
Department of Electrical Engineering, Amirkabir University of Technology, Iran

Abstract: This study presents Intra-Inter Block Control Flow Checking method (I2BCFC) based on software instruction level redundancy for intra-inter block control flow errors detection. In utilizing electronic equipment, space applications designers have two options ahead: radiation-hardening or radiation-tolerant equipment and Commercial Off-The Shelf (COTS) equipment which COTS is the appropriate option due to some reasons like cost and accessibility in many applications. COTS equipment in applications like space missions do not have tolerance capability against some threats such as Single Event Upsets (SEU). Therefore, there should be considerations in resisting these equipment against possible threats. I2BCFC as a technique for COTS reliability increase, has the overhead of 29% in memory in relation to the without code redundancy and this rate has at least 11% improvement in memory overhead in comparison with the previous methods. I2BCFC method has the average performance overheard of 27% in comparison with the normal state of program that it is 5% better than the previous methods.

Keywords: Commercial off-the-shelf, control flow, error detection, radiation-hardening, radiation tolerant, redundancy, single event upset

INTRODUCTION

Space applications for using electronic equipment have two options ahead: rad-hard or rad-tol and COTS equipment. Utilizing rad-hard and rad-tol equipment costs a lot, therefore, using COTS equipment is the appropriate option. Utilizing COTS equipment is now common in many applications such as industrial (Rajabzadeh *et al*., 2004; Rajabzadeh and Miremadi, 2006a; Srinivasan and Lundqvist, 2002; He and Avizienis, 2000; Gill *et al*., 2003), real time (Srinivasan and Lundqvist, 2002; He and Avizienis, 2000; Gill *et al*., 2003; Oh *et al*., 2002a; Chevochot and Puaut, 2001), military (Pizzica, 1998; Trujillo, 1995), avionic industry (Tso and Galaviz, 1999; Newton, 1997; Profeta *et al*., 1996) and space applications (Pignol, 2010; Aicardi *et al*., 2002; Provost *et al*., 2007; Souyris *et al*., 2005; Whisnant *et al*., 2002; Rebaudengo *et al*., 2003; Lovelette *et al*., 2002; Czajkowski and McCartha, 2003; Behr *et al*., 2003; Guidal and David, 1993; Kanai *et al*., 2005; Hihara *et al*., 2003; DeCoursey *et al*., 2006; Hillman *et al*., 2003; Pignol, 2007; Pouget *et al*., 2000; smith and Hengemihle, 1990; Kellner *et al*., 2001; Odenwald and James, 2007).

A large percentage of transient faults in system are converted to latent faults in total output of the system and in other words, their effects are eliminated in the system and are not transferred to the final output. It has been experimentally proved that about 33 to 77% of faults cause Control Flow Errors (CFE) and the remained percentage is converted to data errors (Alkhalifa *et al*., 1999). Therefore, it can be concluded that replacing new techniques for detecting control flow checking errors instead of traditional techniques of transient fault detection in the architecture layer can eliminate additional costs of detecting ineffective faults of system and so improve system efficiency and reduce its cost (Zhu and Aydin, 2006).

Transient fault detection is the first and most important step for making system tolerant to them. Success in this stage can provide appropriate fault coverage for the system. For detecting transient faults and control flow checking errors, some techniques are presented that can be categorized in two general classes of hardware and software redundancy. The methods based on hardware redundancy have a better fault coverage but impose more cost and overhead to the system and do not satisfy users in general purpose applications. Software techniques have less fault coverage and more delay, however, these methods put less cost and overhead on the system and are utilized in different systems due to their flexibility. For program flow checking, the source code is divided into some basic blocks and code running intra the blocks and the branch between them are checked (for example by the watchdog processor). Each basic block is consisted of

Corresponding Author: Hassan Taheri, Department of Electrical Engineering, Amirkabir University of Technology, Iran

some instructions that are located among jump instructions. The errors that should be analyzed in this method are classified into three general categories:

- Wrong occurred jumps intra a basic block
- Wrong occurred jumps inter two basic blocks
- 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, should be able to handle these errors.

In this study, a software-based technique for control flow error detection is suggested that has a better fault coverage beside a much better performance and less memory overhead in relation to other implemented techniques in this field.

In the second section of this study, the previous and related works of hardware and software control flow checking methods are explained. The third section introduces the proposed method. The experimental results of the proposed method are explained in the forth section of this study.

LITERATURE REVIEW

Transient fault occurrence in computer systems and during their running leads to considerable damages. For achieving reliability in computer systems, control flow error 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.

In hardware redundancy methods, a processor and a watchdog timer are utilized to detect control flow errors. In these methods, actual run of the system is compared with the expected run and in case of any contradiction between them; control flow error is detected. Generally in this category of methods, the running flow of the 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 (Jafari-Nodoushan *et al*., 2008).

The first research in this field carried out in 1990. One of the primary methods for detecting transient faults in processors is 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. Figure 1 shows the general structure of a system that has watchdog processor (Kongetira *et al*., 2005).

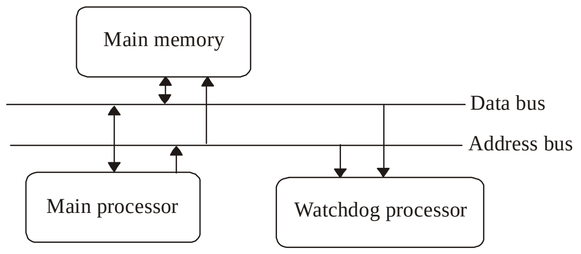

Fig. 1: The structure of a system with watchdog processor

Fault detecting methods based on watchdog processor have two phases. In phase one that is called *initialization phase*, some information about the processor and the process are given to the watchdog. In the second phase, the co-processor monitors the main processor and extracts the related information. Fault detection process is completed when the collected information is compared with the initialization information. The collected information is consisted of memory access mechanism, control flow, control signals and logical results.

The structure of watchdog processor can be applied in system without creating so many changes. Each program is shown by a graph in which nodes show part of the program and edges show the control flow. Each node can be a single statement, a block of branch-free instructions, without-branch interval, or a single procedure. All of the control flow error detection methods are based on the label that is assigned for each node. Watchdog is assigned to the labels and the relationship between labels. During running a program, watchdog checks the flow control of the program, computes nodes labels concurrently and compares with the previous label. Each mismatch or conflict between two labels is assigned as an error. Many methods have been delivered to compare labels that are different in nodes and branch definitions.

In software redundancy methods, the general procedure of the operation is similar to the previous methods and based on the application, there is the possibility of detecting three kinds of control flow errors. The difference between software and hardware methods is that control flow checking of program code is performed by the main processor and instead of imposing hardware redundancy; software redundancy is utilized in the program code. The basis of CFC (Control Flow Checking) methods is on comparing control graph of the running program with the control flow that is predicted statically and at the beginning of the program. Software redundancy methods are usually performed based on data or code copying and can be placed at the procedure level or statement (Li and Hong, 2010; Oh

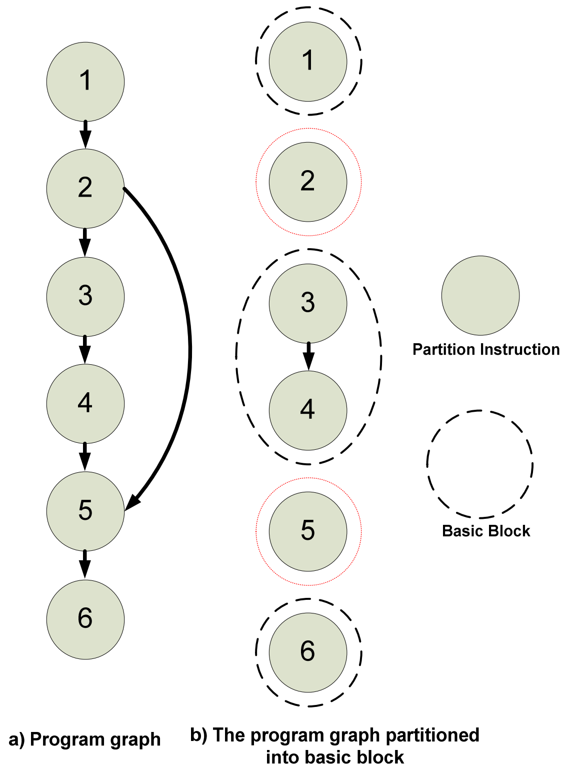

Fig. 2: Program dividing into some basic blocks (Rajabzadeh *et al.*, 2004)

et al., 2002b; Alkhalifa *et al.*, 1999; Jianli *et al.*, 2010; Gadlage *et al.*, 2008). In these methods, some redundant machine-level instructions are added to the program. Moreover, the running program of the processor is divided into some basic blocks that have certain numbers of instructions and are branch-free instructions. For each block, a signature is assigned. During program running, control flow is checked by these signatures till the running is entered in the correct points of program blocks and exited. Figure 2 shows program dividing into basic blocks and partitioning instructions. The program is shown by a direct graph in which each node shows an instruction of the machine and the edges are control flow. Machine instructions are placed consecutively in the main memory. In Fig. 2a, 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 Fig. 2b, the main program is divided into three basic blocks (Rajabzadeh and Miremadi, 2006b).

In some special applications like space systems, due to beam and heavy ion radiation, many transient faults lead to disorder in flow control and software systems and therefore unpredictable behavior. 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 Single Event Upset (SEU). Many studies have worked on the crash of neutrons, protons and electrons that lead to SEU (Gadlage *et al.*, 2008; Felix *et al.*, 2008; Kruckmeyer *et al.*, 2008; Shaneyfelt *et al.*, 2008; Maestro and Reviriego, 2009; Petersen, 2008).

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 equipment. One of the works in this field is RSCFC (Li and Hong, 2010) in which the program is divided into some basic blocks. In the first stage, 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 of the program are detected by adding runtime signatures with the information at the beginning and end of the blocks.

With comparison to the previous works, this method has more fault coverage and a better efficiency and it also consumes less memory (Li and Hong, 2010). In RSCFC method, if the program is divided to n basic blocks, then the assigned signature to each block should be n+1 bit in that the most significant bit should have the value of 1. Each bit of this signature except from the most significant bit is for one block (Oh *et al.*, 2002a).

In CFCSS (Oh *et al.*, 2002a) method a signature of *s* and a signature of *d* (the last basic block signature XORed with *s*) are assigned to each basic block. A global variable of *G* is also added to the program which is consisted of running block signature amount. During the program running, whenever the program running enters to a new basic block, *G* is updated to a new amount.

ECCA (Alkhalifa *et al.*, 1999), that is another CFC method, is implemented in high level of RTL. In high level implementation, ECCA adds two instructions and an ID, a prime number to each basic block. This technique is implemented by manipulating GCC compiler.

THE PROPOSED TECHNIQUE

In the previous section, some of the presented techniques for control flow error detection based on software and hardware were analyzed. The presented technique of this study is delivered in the

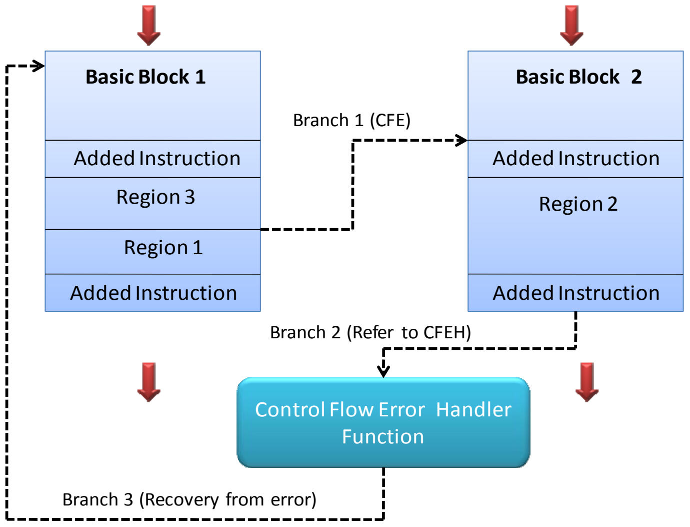

Fig. 3: The way of detection and correction of flow control errors in processors

following. With regarding to the mentioned advantages of the methods based on the software, the presented technique in this study 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.

Figure 3 shows a control flow error between two basic blocks in that the running of a code segment inside the basic block number 1 is incomplete due to the occurrence of a flow control error and the running is mistakenly transferred to the basic block number 2. At the end of basic block number 2, this error is detected and the control is transferred to the function that occurred to correct this control flow error. This function has the capability of detecting the basic block in which the flow control error happened. It transfers the control to the beginning of that basic block.

In the presented technique of this study, for detecting control flow errors, a signature is assigned to each basic block. This signature is variable s_i that shows successor blocks of the present block.

For control flow checking in basic blocks, three instructions are defined. The first instruction 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 controlling the correctness of the accessibility, Eq. (1) is utilized:

$$err = s\,[sel] \tag{1}$$

s is s_i variable that is updated by running the program. At the middle of each basic block, if *sel* number of the bit (this 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 and the program control is returned to the beginning of the *check* instruction of the previous node and is run again there.

Another instruction is called *update* and is run to update s signature. For updating flow control signature, Eq. (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 s is set to 00000…1 for the first time to be just able to go to the first basic block and other jumps become impermissible for it.

The third instruction is called *exit* that is run at the time of exiting the basic block and takes the amount of *sel* variable to the number that is the sign of the present basic block.

Two initial instructions are placed in the middle of each basic block and so some of the errors caused by impermissible interior jumps in a basic block are detected. After detecting error, *err signal* equals 0 and the program control is returned to the beginning of check instruction in source block and running is started gain.
Therefore, there is a high probability that by further running, the error is corrected. If the returning action to a specified place is more than the threshold limit, it is suggested to reset the whole system to remove this problem. *Exit* instruction is placed at the end of each basic block. Figure 4 shows the implementation of the mentioned method for bubble sort.

```
//basic block 1
for (int pass=1; pass < n; pass++) {
//basic block 2
    for (int i=0; i < n-pass; i++) {
        if (x[i] > x[i+1]) {
//basic block 3
            int temp = x[i];  x[i] = x[i+1];  x[i+1] = temp;
        }
//basic block 4
    }
//basic block 5
                                }
```

Fig. 4: Bubble sort sample program code and the way of its distribution to basic blocks

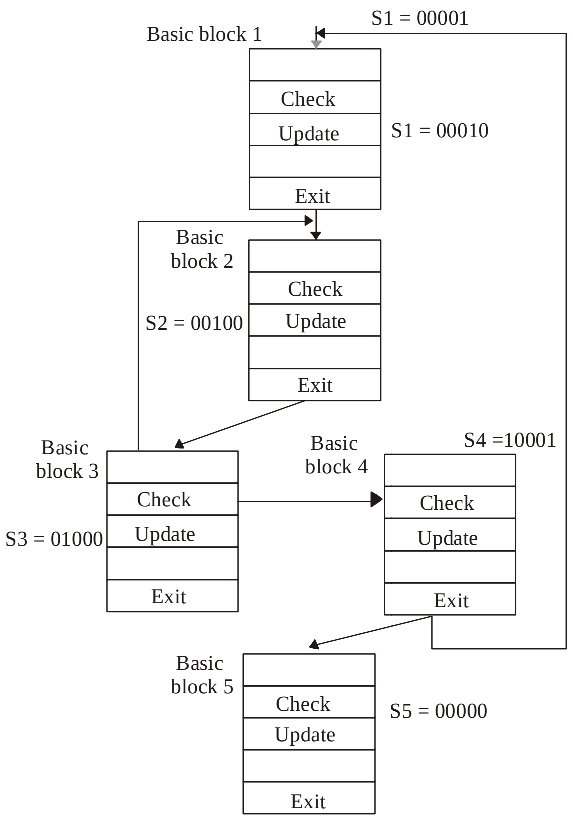

Fig. 5: Control graph of sample bubble sort program

Control graph of the mentioned program is shown in Fig. 5.

In control graph of Fig. 5, basic blocks interconnections are shown. According to this figure, a signature is assigned for each basic block in which successor blocks for the present block are specified. In this graph, basic blocks interconnections and their interactions are also specified. The interior structure of a basic block by inserting the redundant instructions is shown in Fig. 6.

By inserting check and update instructions, all of the single errors due to incorrect jumps can be found. The proofs of this claim are as follows:

One jump from *vi* node to *vj* node ($Vj \in succ(Vi)$) that is shown in Fig. 7.

The first state: when an impermissible and unwanted jump occurs at the end of *vi* block to the middle of *vj* block.

In this state, when check instruction is run in *vj* block, since *sel* variable amount in *vj* block is not updated, *err* variable is equaled 0 and this error is detected.

For example, imagine in Fig. 7, there is an unwanted jump from the end of the first basic block to

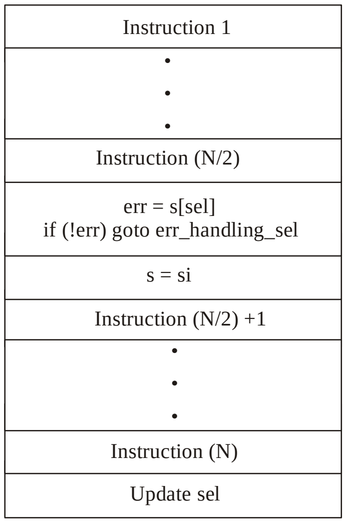

Fig. 6: The interior structure of a basic block by inserting redundant instructions

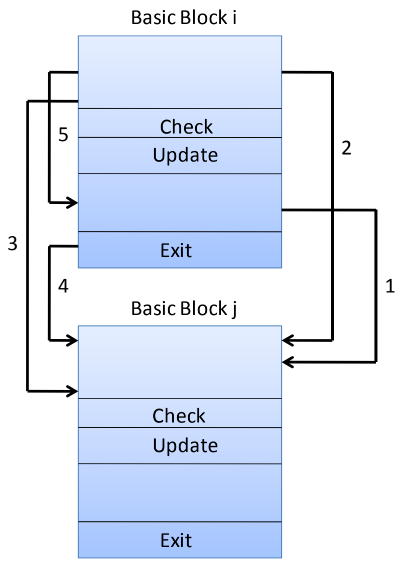

Fig. 7: Impermissible jumps detectable by the presented technique in case $v_j \in succ(v_i)$

the middle of the second block. In this case, since sel variable at the end of the first block is not updated, has the amount of zero. When the operation is reached to check and update instructions of the second basic block, number 0 bit from *s* variable that now has signature-BB1 amount (equal to 0) is assigned to err signal. The err signal receives the amount of 1 and error is detected. In this case:

s = signature_BB1 = 00010
s [0] = 0
err = s [0] = 0

The second state: an impermissible jump occurs from the middle of *vi* basic block to the beginning of *vj* basic block.

In this state, since redundant instructions in *vi* block is not run, so *sel* variable is not updated and the error is detected like the first state.

For example, imagine in Fig. 7, there is an unwanted jump from the middle of first block to the beginning of the second block. This example is totally like the first state in which *sel* variable has the amount of zero at the time of entrance to the second basic block and like the last example; the error is detected in check instruction of the second basic block.

The third state: an impermissible jump occurs from the middle of *vi* block to the middle of *vj* block.

In this state like the previous states, the error is detected in the next check instruction due to not running of exit instruction.

For example, imagine in Fig. 7, there is an unwanted jump from the middle of first block to the middle of the second block. Error detection in this state is completely the same as the mentioned cases in the first and second states.

The fourth state: an impermissible jump occurs from the end of *vi* block to the beginning of *vj* block.

In this state, like the other ones, due to not running of exit instruction, an error is detected in the next check instruction.

For example, imagine in Fig. 7, there is an unwanted jump from the end of first block to the beginning of the second block. Error detection in this state is also similar to the previous states.

A jump from *vi* node to *vj* node in that $Vj \,!\!\in succ\ (Vi\,)$ occurs impermissibly.

The fifth state: when an impermissible and unwanted jump occurs from the end *vi* block to the beginning of *vj* block. It is like the fourth state. In this state, like the other states, error is detected due to not running of exit instruction.

For example, imagine in Fig. 7, there is a jump from the end of the first block to the beginning of the third block. In check instruction of the third block, the bit 0 of *s* is checked. Since *sel* is not updated due to the unwanted jump and since this bit is 0 in *s* or *si*, *err* signal equals 1 and the error is detected.

The sixth state: when an impermissible and unwanted jump occurs from the end of *vi* block to the middle *vj* block. It is similar to the first state.

In this state, like the other states, error is detected due to not running of exit instruction. For example, imagine in Fig. 7, there is a jump from the end of the first block to the middle of the third block. In this state, similarly to the fifth state, *err* signal in the third block equals 1 and error is detected.

The seventh state: when an impermissible and unwanted jump occurs from the middle of *vi* block to the beginning of the *vj* block. It is similar to the second state.

In this state, like the other states, error is detected duo to not running of exit instruction. For example, in Fig. 7, imagine there is a jump from the middle of the first block to the beginning of the third block. This state is also similar to the two previous states.

The eighth state: when an impermissible and unwanted jump occurs from the middle of *vi* block to the middle of *vj* block. It is similar to the third state. In this state, like the other states, error is detected due to not running of exit instruction.

For example, imagine in Fig. 7, there is a jump from the middle of the first block to the middle of the third block. In check instruction of the third block, the 0 bit of *s* is checked. Since *sel* is not updated due to the unwanted jump and this bit in *s* or *si* is zero, *err* signal equals 1 and error is detected.

Further to the mentioned states, the presented method of this study has the capability of detecting some of the errors that occur due to impermissible jump in a basic block.

When an unwanted jump occurs from one instruction in *vi* basic block to another instruction of the same block. This state is shown in Fig. 7 and with the arrow 5.

The ninth state: when an unwanted jump occurs from the instructions before check and update to the instructions after them.

In this state, since *s* variable has not been updated during running, error is detected in check instruction of the next block.

For example, imagine an unwanted jump occurs from before check instruction of the first block to after update instruction of the same block. In this state, due to the fact that update instruction has not been run, *s* variable takes its initial amount that is 00001. At the end of the first block, *sel* variable amount is updated to 1 and the program enters the second basic block. In the second basic block, in check instruction, the first bit of S variable is assigned to *err* and it leads to detecting the occurred error:

$$s = s_{initial} = 00001,\ err = s\,[sel] = 0$$

EXPERIMENTAL RESULTS

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

Test environment: For analyzing the proposed method, the infrastructure shown in Fig. 8 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 (Asghari *et al*., 2010)
- Development board phyCORE-MPC555
- A personal computer

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

- Direct fault injection onto processor registers by the 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

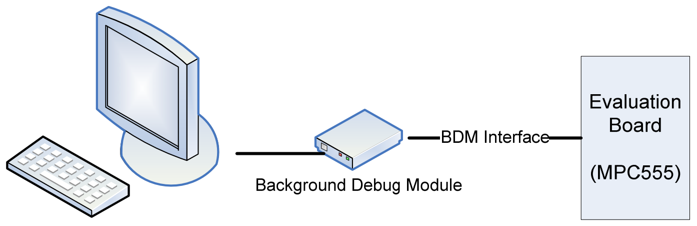

Fig. 8: Fault injection mechanism structure by the use of BDM

Table 1: Fault injection results in comparison with CFCSS, ECCA and RSCFC

	CR (%)	OS (%)	WR (%)	TO (%)	SD (%)
BS-CFCSS	25.65	39.88	8.80	2.87	22.80
MM-CFCSS	14.50	43.87	12.14	6.15	23.34
QS-CFCSS	7.13	40.20	8.40	7.96	36.31
BS-ECCA	16.05	43.00	10.25	8.10	22.60
MM-ECCA	12.70	41.09	11.20	5.87	29.14
QS-ECCA	10.70	36.66	10.54	4.30	37.80
BS-RSCFC	12.30	46.10	11.60	7.50	22.50
MM-RSCFC	10.96	42.34	9.70	2.50	34.50
QS-RSCFC	14.09	40.65	11.20	6.56	27.50
BS-I2BCFC	12.80	40.43	10.23	6.99	29.55
MM-I2BCFC	16.60	29.70	11.80	4.34	37.56
QS-I2BCFC	17.10	28.10	9.90	5.30	39.70

of Bubble Sort (BS), Quick Sort (QS) and 40×40 Matrixes Multiplication (MM) that 5000 faults are injected on them. Since in the method of direct fault injection onto processor registers by the use of BDM, processor registers can be directly manipulated, 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 Asghari *et al*. (2010), 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 the reality. As a result, the second and third methods are utilized for analyzing error detection method.

For analyzing the proposed method, fault injection methods suggested in section 4-1 are utilized. Table 1 shows the results of fault injection (CR: Correct Result, OS: Operating System, WR: Wrong Result, TO: Time Out and SD: Single Detection). RSCFC 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 control flow error 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.
In the proposed idea of this study, only three 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. It is clear that memory and performance overhead of the presented technique in this study is much less than RSCFC technique. In reference (Li and Hong, 2010), RSCFC memory and performance overhead are compared with CFCSS and ECCA and is claimed that less overhead has been consumed in RSCFC.

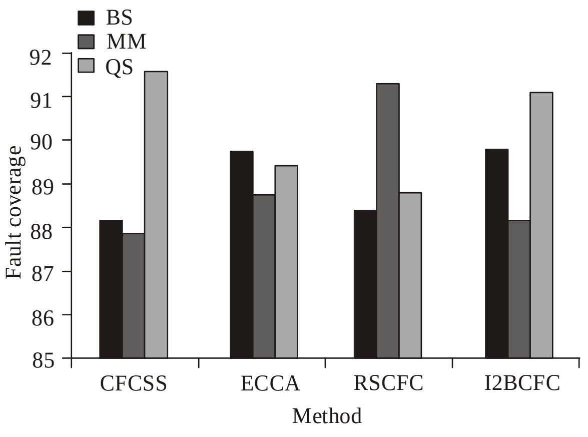

Fig. 9: Fault coverage comparison for BS, MM and QS benchmarks

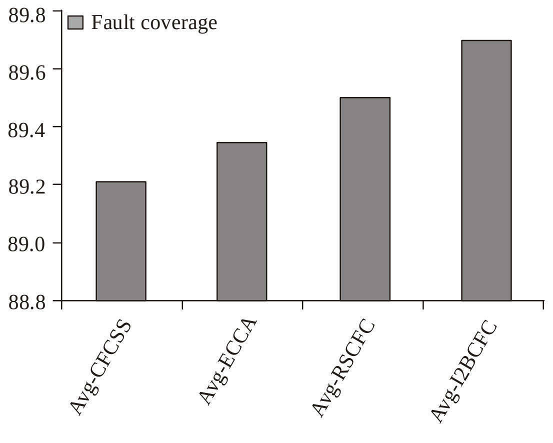

Fig. 10: Total fault coverage comparison

Therefore, it is reasonable that the overhead of the proposed technique in this study is less than other techniques.

By comparing the size and speed of program running in which CFCSS, ECCA and RSCFC methods are applied, Fig. 9, 10 and 11 results are achieved that show I2BCFC technique has the overhead of 1.29 in

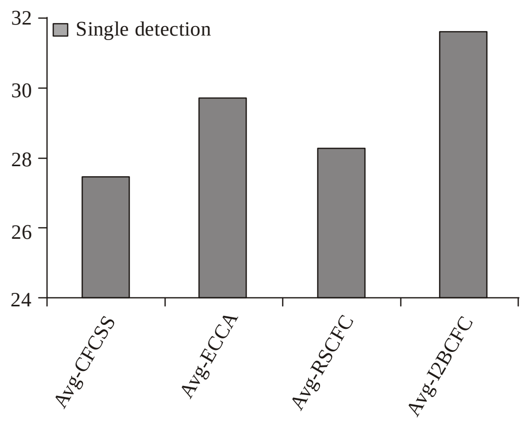

Fig. 11: Single error detection comparison

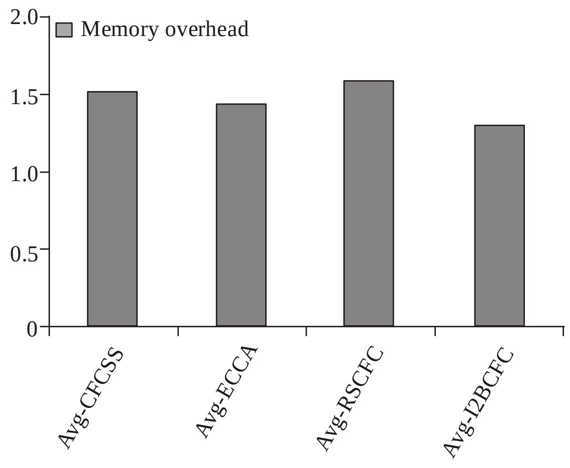

Fig. 12: Memory overhead comparison

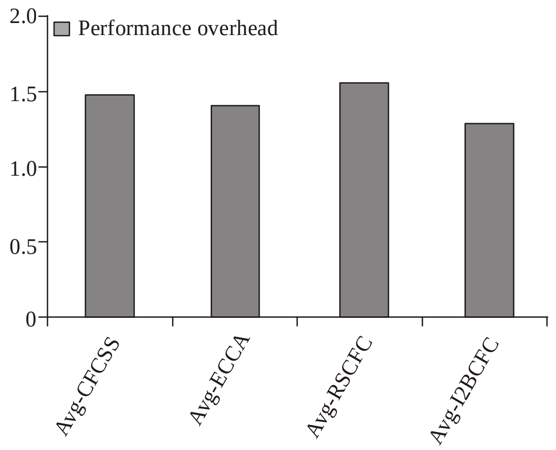

Fig. 13: Performance overhead comparison

memory in relation to the without code redundancy and this rate has at least 11% of improvement in memory overhead in comparison with the previous methods. I2BCFC method has the average performance overheard of 1.27% in comparison with the normal state of program which is 5% better than the previous methods. I2BCFC technique has the average single fault coverage of 34% and in comparison with the previous methods has at least 3% improvement.

Figure 12 and 13 show that the proposed method not only has a better fault coverage, but also delivers a much less memory and performance overhead in comparison with other implemented techniques and; therefore, its usage is more appropriate.

Analytical computation of fault coverage: The following states are for analyzing impermissible jumps that lead to control flow errors:

- **The jump from one basic block to another:** In this state a jump happens due to the fault occurrence in program counter from one basic block to another basic block that is not a permissible destination. The probability of this occurrence is like being in the *i* basic block and after a fault occurrence there is a jump from the *i* basic block to *j* basic block and no failure happens for the program due to this jump occurrence. The probability of this kind of error occurrence is computed by the Eq. (3):

$$P_{type\text{-}1} = P_{BBi} * P_{BBi \longrightarrow BBj} * (1\text{-}P_{program\ crash}) \quad (3)$$

- **The jump from one basic block to the outside of the program:** In this state a jump happens due to the fault occurrence in program counter from one basic block to a place outside the program. The probability of this occurrence is like being in the *i* basic block and after a fault occurrence there is a jump from the *i* basic block to a place outside the program and no failure happens for the program due to this jump occurrence. The probability of this kind of error occurrence is computed by the Eq. (4) (PB in this equation is Partition Block):

$$P_{type\text{-}2} = P_{BBi} * P_{BBi \longrightarrow PB} * (1\text{-}P_{program\ crash}) \quad (4)$$

- **The jump from one basic to itself:** In this state a jump happens due to the fault occurrence in program counter from one basic block to another place in the same block that is not a permissible destination. The probability of this occurrence is like being in the *i* basic block and after a fault occurrence there is a jump from the *i* basic block to a place in the same basic block and no failure happens for the program due to this jump occurrence. The probability of this kind of error occurrence is computed by the Eq. (5):

$$P_{type\text{-}3} = P_{BBi} * P_{BBi \longrightarrow BBi} * (1\text{-}P_{program\ crash}) \qquad (5)$$

Type 3 jump is consisted of two kinds. Each basic block is divided into two parts: The upper part that is before redundant instructions of each basic block and the lower part that is the section of each basic block after redundant instructions. Two jumps happen due to these parts:

- A jump from the upper part of a basic block to the lower part of it and vice versa. The probability of this state can be computed by the Eq. (6):

$$P_{type\text{-}3\text{-}1} = P_{BBi} * (P_{BBiu \longrightarrow BBiD} + P_{BBiD \longrightarrow BBiu}) * (1\text{-}P_{program\ crash}) \qquad (6)$$

- A jump from the upper part of a basic block to another place of the upper part or a jump from the lower part of a basic block to another place of the lower part. The probability of this state can be computed by the Eq. (7):

$$P_{type\text{-}3\text{-}2} = P_{BBi} * (P_{BBiu \longrightarrow BBiu} + P_{BBiD \longrightarrow BBiD}) * (1\text{-}P_{program\ crash}) \qquad (7)$$

Generally, the probability of jumps in each basic block can be restated as Eq. (8):

$$P_{BBi \longrightarrow BBi} = P_{BBiu \longrightarrow BBiu} + P_{BBiD \longrightarrow BBiD} + P_{BBiu \longrightarrow BBiD} + P_{BBiD \longrightarrow BBiu} \qquad (8)$$

Since:

$$P_{BBiu \longrightarrow BBiu}(x, y) = P(P_{BBiu \longrightarrow BBiu} \mid z = 1)\ P(z = 1) + = P(P_{BBiu \longrightarrow BBiu} \mid z = 0)\ P(z = 0)$$

In that *z* variable has Bernoulli distribution, therefore:

$$P(z = 0) = P(z = 1) = ½$$

For maintaining in the upper part of the basic block the following conditions should be met:

$$P(P_{BBiu \longrightarrow BBiu} \mid z = 1) = P(L(x) <= y)$$
$$P(P_{BBiu \longrightarrow BBiu} \mid z = 0) = P(L(x) <= S_{BB}/2\text{-}y)$$

In the above equations, z is a Bernoulli random variable that shows jump direction. S_{BB} variable shows the average size of a basic block based on byte and is computed by multiplying the average length of program instructions and the average number of instructions of each basic block. The *y* random variable has uniform distribution and determines the distance of fault occurrence place from the beginning of the basic block based on byte.

SEU errors in program counter lead to control flow errors. The *x* random variable has the amount of 0 to 31 and shows the erroneous bit number in the program counter. The *L(x)* variable has the amount of 2^x and shows how many impermissible jumps happen by changing *x* in program counter.

For computing jump probability of each basic block to another basic block, the Eq. (9) is utilized (in this equation, I_{av} is average number of instruction bytes):

$$P_{BBi \longrightarrow BBj}(x) = 1/N_{BB} * 1/2 * 1/N_B \sum_{k=0}^{N_{BB}} \sum_{i=1}^{N_B} \sum_{j=1}^{i-1} P(I_{av}.k + j.S_{PB} + (j\text{-}1)S_{BB} < L(x) < I_{av}.k + j.S_{PB} + j.S_{BB}) + 1/N_{BB} * 1/2 * 1/N_B \sum_{k=0}^{N_{BB}} \sum_{i=1}^{N_B} \sum_{j=1}^{N_B-i} P(S_{BB}\text{-}I_{av}.k + j.S_{PB} + (j\text{-}1)\, S_{BB} < L(x) < S_{BB}\text{-}I_{av}.k + j.S_{PB} + j.\, S_{BB}) \qquad (9)$$

The jump probability from one basic block to a place outside the program is computed by the Eq. (10):

$$P_{BBi \longrightarrow BBj}(x) = 1/N_{BB} * 1/2 * 1/N_B \sum_{k=0}^{N_{BB}} \sum_{i=1}^{N_B} \sum_{j=1}^{i-1} P(I_{av}.k + (j\text{-}1).S_{PB} + (j\text{-}1)\, S_{BB} < L(x) < I_{av}.k + j\, S_{PB} + (j\text{-}1).S_{BB}) + 1/N_{BB} * 1/2 * 1/N_B \sum_{k=0}^{N_{BB}} \sum_{i=1}^{N_B} \sum_{j=1}^{N_B-i} P(S_{BB}\text{-}I_{av}.k + (j\text{-}1).S_{PB} + (j\text{-}1)\, S_{BB} < L(x) < S_{BB}\text{-}I_{av}.k + j.S_{PB} + (j\text{-}1).\, S_{BB}) \qquad (10)$$

I2BCFC method has a better fault coverage in comparison with the previous methods due to detecting a percentage of first kind of type 3 jump. The Table 2 shows the results of applying I2BCFC method on Matrix Multiplying, Quick Sort and Bubble Sort benchmarks.

Table 2: Results of applying I2BCFC method on three benchmarks

Benchmarks	$P_{type\text{-}1}$	$P_{type\text{-}2}$	$P_{type\text{-}3}$*
QS	74.70	15.70	9.60
MM	72.10	16.70	11.20
BS	78.60	12.60	8.80
Average	75.13	15.00	9.86

*: $P_{type\ 3\text{-}1(QS)}$: 4.4; $P_{type\ 3\text{-}1(MM)}$: 5.3; $P_{type\ 3\text{-}1(BS)}$: 3.3; $P_{type\ 3\text{-}1(Average)}$: 4.33; $P_{type\ 3\text{-}2(QS)}$: 5.2; $P_{type\ 3\text{-}2(MM)}$: 5.9; $P_{type\ 3\text{-}2(BS)}$: 5.5; $P_{type\ 3\text{-}1(Average)}$: 5.53

CONCLUSION

Utilizing COTS equipment is one of the appropriate choices in the 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 have lots of dangers. In this study, a method based on software redundancy in instruction level has been delivered that has a better performance and less memory overhead in comparison with the previous presented methods. By utilizing this method, a great percent of the single transient faults that lead to control flow errors can be detected.

ACKNOWLEDGMENT

This study was supported by the Research Institute for ICT-ITRC.

REFERENCES

Aicardi, C., P. Lay, A. Mouton, C. Revellat and D. Beauvallet *et al*., 2002. Guidelines for commercial parts management. Proceeding of European Space Components Conference (ESCCON), 24-27 September 2002, Toulouse, France, pp: 185-188.

Alkhalifa, Z., V.S.S. Nair, N. Krishnamurthy and J.A. Abraham, 1999. Design and evaluation of system-level checks for on-line control flow error detection. IEEE Trans. Parall. Distr. Syst., 10(6): 627-641.

Asghari, S.A., H. Pedram, H. Taheri and M. Khademi, 2010. A New Background Debug Mode Based Technique for Fault Injection in Embedded Systems. Int. Rev. Model. Simul. (IREMOS), 3(3): 415-422.

Behr, P., W. Bärwald, K. Briess and S. Montenegro, 2003. Fault tolerance and COTS: Next generation of high performance satellite computers. Proceeding of Eurospace DAta Systems In proceeding of Aerospace conference (DASIA), 2-6 June 2003, CDROM, Prague, pp: 76-76.

Chevochot, P. and I. Puaut, 2001. Experimental evaluation of the fail silent behavior of a distributed real-time run-time support built from COTS components. Proceedings of the International Conference on Dependable Systems and Networks (DSN-2001), IEEE Computer Society Washington, DC, USA, pp: 304-313.

Czajkowski, D. and M. McCartha, 2003. Ultra low-power space computer leveraging embedded seu afitigation. Proceeding of IEEE Aerospace conference, Space Micro, 5: 2315-2328.

DeCoursey, R., R. Melton and R. Estes, 2006. Non radiation hardened microprocessors in space-based remote sensing systems. Proc. SPIE. 6361: 63611-63611.

Felix, J.A., J.R. Schwank, M.R. Shaneyfelt, J. Baggio, P. Paillet, V. Ferlet-Cavrois, P.E. Dodd, S. Girard and E.W. Blackmore, 2008. Test Procedures for Proton-Induced Single Event Latchup in Space Environments. IEEE Trans. Nucl. Sci., 55(4): 2161-2165.

Gadlage, M.J., R.D. Schrimpf, B. Narasimham, J.A. Pellish, K.M. Warren, R.A. Reed, R.A. Weller, B.L. Bhuva, L.W. Massengill and X. Zhu, 2008. Assessing Alpha Particle-Induced Single Event Transient Vulnerability in a 90-nm CMOS Technology. IEEE Electr. Device Lett., 29(6): 638-640.

Gill, C.D., R.K. Cytron and D.C. Schmidt, 2003. Multi paradigm scheduling for distributed real-time embedded computing. Proceeding of IEEE Special Issue on Modeling and Design of Embedded Systems, 91(1): 183-97.

Guidal, C. and P. David, 1993. Development of a fault tolerant computer system for the HERMES space shuttle. The Twenty-Third International Symposium on Fault-Tolerant Computing, 1993. FTCS-23. Digest of Papers, Matra Marconi Space, Toulouse, France, pp: 641-646.

He, Y. and A. Avizienis, 2000. Assessment of the applicability of COTS microprocessors in high-confidence computing systems: A case study. Proceedings of the international conference on dependable systems and networks (DSN-2000), Dept. of Comput. Sci., California Univ., Los Angeles, CA, pp: 81-86.

Hihara, H., K. Yamada, M. Adachi, K. Mitani, M. Akiyama and K. Hama, 2003. CRAFT: An experimental fault tolerant computer for SERVIS-2 satellite. Proceeding of 21st AIAA International Communications Satellite Systems Conference and Exhibit, AIAA 2003-2291.

Hillman, R., G. Swift, P. Layton, M. Conrad, C. Thibodeau and F. Irom, 2003. Space processor radiation mitigation and validation techniques for an 1,800 MIPS processor board. Proceeding of 12th RADECS Association / ESA / IEEE European Conference on Radiations and its Effects on Components and Systems (RADECS), pp: 347-352.

Jafari-Nodoushan, M., G. Miremadi and A. Ejlali, 2008. Control-flow checking using branch instructions. Proceeding of the 8th International Conference on Embedded and Ubiquitous Computing, 17-20 Dec. 2008, Dependable Syst. Lab., Sharif Univ. of Technol., pp: 66-72.

Jianli, L., T. Qingping and X. Jianjun, 2010. A software-implemented configurable control flow checking method. Proceeding of International Symposium on Parallel Architectures, Algorithms and Programming (PAAP), IEEE Computer Society Washington, DC, USA, pp: 199-205.

Kanai, H., K. Hama, M. Akiyama and N. Natsuka, 2005. Overview of SERVIS project toward application of commercial technology for space. Proceeding of 56th International Astronautically Congress, IAC-05-D1.2.09.

Kellner, A., H.J. Kolinowitz and G. Urban, 2001. A novel approach to fault tolerant computing. Proceeding of IEEE Aerospace conference, Space Infrastructure Div., Astrium GmbH, Bremen, 3: 1127-1131.

Kongetira, P., K. Aingaran and K. Olukotun, 2005. Niagara: A 32-Way multithreaded sparc processor. IEEE Micro, 25(2): 21-29.

Kruckmeyer, K., R.L. Rennie and V. Ramachandran, 2008. Use of Code Error and Beat Frequency Test Method to Identify Single Event Upset Sensitive Circuits in a 1 GHz Analog to Digital Converter. IEEE Trans. Nucl. Sci., 55(4): 113-117.

Li, A. and B. Hong, 2010. On-line control flow error detection using relationship signatures among basic blocks. Comput. Electr. Eng., 36(1): 132-141.

Lovelette, M.N., K.S. Wood, D.L. Wood, J.H. Beall and P.P. Shirvani *et al*., 2002. Strategies for fault-tolerant, space-based computing: lessons learned from the ARGOS testbed. Proceeding of IEEE Aerospace conference, Naval Res. Lab., Washington, DC, USA, 5: 2109-2119.

Maestro, J.A. and P. Reviriego, 2009. Reliability of Single-Error Correction Protected Memories, IEEE Trans. Reliab., 58(1): 193-201.

Newton, C., 1997. Design assurance for airborne COTS hardware. Proceeding of IEE colloquium on COTS and safety critical systems (Digest No. 1997/013), 28 Jan 1997, Defence Res. Agency, Malvern, pp: 4/1-3.

Odenwald, S. and C. James, 2007. NASA, Space Math-III, Hinode satellite program's Education and Public Outreach Project.

Oh, N., P.P. Shirvani and E.J. McClusky, 2002a. Control Flow Checking By Software Signature. IEEE Trans. Reliab., 5(2): 111-122.

Oh, N., PP. Shirvani and E.J. McCluskey, 2002b. Error detection by duplicated instructions in super-scalar processors. IEEE Trans. Reliab., 51(1): 63-75.

Petersen, E.L., 2008. Single-Event Data Analysis. IEEE Trans. Nucl. Sci., 55(6): 2819-2841.

Pignol, M., 2007. Methodology and tools developed for validation of COTS based fault-tolerant spacecraft supercomputers. Proceeding of 13th IEEE International on-Line Testing Symposium, 8-11 July 2007, CNES, Toulouse, pp: 85-92.

Pignol, M., 2010. COTS-based Applications in Space Avionics. Design, Automation and Test in Europe Conference and Exhibition (DATE), 8-12 March 2010, CNES, Toulouse, France, pp: 1213-1219.

Pizzica, S.V., 1998. Meeting military system test requirements with the usage of COTS products. Proceedings of the 17th AIAA/IEEE/SAE digital avionics systems conference (DASC), 31 Oct-7 Nov 1998, Raytheon Syst. Co., El Segundo, CA, pp: B45/1-7.

Pouget, V., P. Fouillat, D. Lewis, H. Lapuyade and L. Sarger *et al*., 2000. An overview of the applications of a pulsed laser system for SEU testing. Proceeding of 6th IEEE Int. On-Line Testing Workshop, IXL, Bordeaux I Univ., Talence, pp: 52-57.

Profeta, J., N. Andrianos, Y. Bing, B. Johnson, T. DeLong and D. Guaspart, 1996. Safety-critical systems built with COTS. Comput., 29(11): 54-60.

Provost, S., M. Le Roy, B. Mamdy, G. Flandin and T. Paulsen, 2007. GAIA video processing embedded algorithms: prototyping and validation activities. Proceeding of Eurospace DAta Systems. Proceeding of Aerospace Conference (DASIA).

Rajabzadeh, A., G. Miremadi and M. Mohandespour, 2004. Error detection enhancement in COTS superscalar processors with performance monitoring features. J. Elec. Test. Theory Appl., 20(5): 553-67.

Rajabzadeh, A. and G. Miremadi, 2006a. CFCET: A hardware based control flow checking technique in COTS processors using execution training. Els. J. Comput. Microelectron. Reliab., 46(5-6): 959-972.

Rajabzadeh, A. and G. Miremadi, 2006b. Transient detection in COTS processors using software approach. Els. J. Microelectron. Reliab., 46(1): 124-133.

Rebaudengo, M., M.S. Reorda and M. Violante, 2003. A new software based technique for low-cost fault-tolerant application. Proceeding of IEEE Reliability and Maintainability Symposium, Politecnico di Torino, Italy, pp: 25-28.

Rujillo, E., 1995. Military requirements constrain COTS utilization. Proceedings of the 14th digital avionics systems conference (DASC), 5-9 Nov 1995, Hughes Aircraft Co., Los Angeles, CA, pp: 112-117.

Shaneyfelt, M.R., J.R. Schwank, P.E. Dodd and J.A. Felix, 2008. Total Ionizing Dose and Single Event Effects Hardness Assurance Qualification Issues for Microelectronics. IEEE Trans. Nucl. Sci., 55(4): 1926-1946.

Smith, B.S. and J. Hengemihle, 1990. The small explorer data system, a data system based on standard interfaces. Proceeding of AIAA/NASA 2nd International Symposium on Space Information Systems. Sept. 17-19, 1990, Pasadena, CA, pp: 9-9.

Souyris, J., E.L. Pavec, G. Himbert, V. Jégu and G. Borios *et al.*, 2005. Computing the worst case execution time of an avionics program by abstract interpretation. Proceeding of 5th International Workshop on Worst-Case Execution Time (WCET) Analysis, pp: 21-24.

Srinivasan, J. and K. Lundqvist, 2002. Real-time architecture analysis: A COTS perspective. Proceedings of the 21st Digital Avionics Systems, Software Eng. Res. Lab., MIT, Cambridge, MA, USA, pp: 5D4-1-9.

Tso, P. and P. Galaviz, 1999. Improved aircraft readiness through COTS. Proceeding of IEEE Systems Readiness Technology Conference (AUTOTESTCON_99), pp: 451-456.

Whisnant, K., R.K. Iyer, P. Jones, R. Some and D. Rennels, 2002. An experimental evaluation of the REE SIFT environment for space borne applications. Proceeding of IEEE/IFIP International Conference on Dependable Systems and Networks (DSN), June 23-June 26, Washington, D.C., USA, pp: 585-594.

Zhu, D. and H. Aydin, 2006. Reliability Effects of Process and Thread Redundancy on Chip Multiprocessors. Proceeding of the 36th Annual IEEE/IFIP International Conference on Dependable Systems and Networks, Department of Computer Science, George Mason, pp: 1-2.