Software-based Control Flow Checking against Transient Faults in Industrial Environments

Seyyed Amir Asghari, Hassan Taheri, Hossein Pedram, and Okyay Kaynak, *Fellow, IEEE*

Abstract— **Mechatronic systems operating in industrial environments are subject to a variety of threats because of harsh conditions. Industrial systems usually use Commercial Off-The Shelf (COTS) equipment which are not robust and safe against hostile conditions and therefore require fault tolerance considerations. This paper presents a novel and efficient method for online detection of control flow errors, called *Software-based Control Flow Checking (SCFC)*. It is implemented purely in software and does not manipulate the hardware architecture of the system. Redundant instructions and signatures are embedded into the program at compile time and are utilized for control flow checking at run time. The signatures of the basic blocks are derived from the program graph. It is shown in the paper that SCFC method can increase single detection capability to 14.7% and the fault coverage to 6.12% averagely in comparison with other methods without any increase in memory and performance overheads. In the paper, besides experimental evaluations, analytical evaluations are also carried out, based on probability principles. The detection ability of each method used is thus computed. These computations verify the experimental results and show that SCFC can detect more errors than other methods suggested in literature. Considering the memory limitations in some (such as space) applications and the trend towards the requirement for faster execution of programs, we suggest a novel metric; called fitness parameter which incorporates these. It is a better measure than the previously proposed ones since it considers the fault coverage, the memory overhead and the execution time (performance overhead) of each method simultaneously, as well as the detection capability.**

Index Terms— **Commercial Off-The-Shelf, control flow checking, fitness parameter, fault injection, analytical evaluation, software-based error detection.**

I. Introduction

IN the selections of the electronic equipment to be used in industrial environments there are two options: to go for specially designed, high reliability but costly or the Commercial Off-The Shelf (COTS) equipment. Utilizing robust equipment can be prohibitively costly; therefore, the use of COTS equipment is the appropriate option in most applications [1-13].

It has been experimentally shown that about 33 to 77 percent of the transient faults cause control flow errors (CFE) and the remaining are converted into data errors [14]. It can therefore be concluded that by the use of new techniques based on control flow checking, instead of the traditional techniques of transient fault detection in the application layer, the additional costs of detecting the faults that will finally be ineffective can be avoided. The system efficiency and its cost can thus be reduced [14-17].

For detecting transient faults some techniques are suggested in literature that would fall into two general classes, hardware or software redundancy. The methods based on hardware redundancy have a better fault coverage but impose higher costs and overheads on the system and therefore may not meet the requirements of some general purpose applications. Software-based techniques have less fault coverage and larger delay; however, they mean lower cost and overhead on the system and can be utilized in different types of COTS systems due to their flexibility. Another point to be considered in comparison with hardware-based methods is that in software-based methods, there is no dependency on hardware or no need for its reconfiguration [15-17].

For control flow checking, the general approach adopted is that the source code is divided into some basic blocks and the code running inside the blocks and the branches between them is checked (for example by a watchdog processor). Each basic block consists of some instructions that are located among jump instructions. Errors that should be analyzed in these methods are classified into three general categories:

- ✓ Illegal jumps intra basic blocks
- ✓ Illegal jumps inter basic blocks
- ✓ Illegal jumps from a basic block to the unused space of the memory

These illegal jumps lead to control flow errors that can be grouped into the following categories:

Type 1: an error caused by an illegal jump from the end of a basic block to the beginning of another basic block,

Type 2: an error caused by a legal but incorrect jump from the end of a basic block to the beginning of another basic block,

Type 3: an error caused by a jump from the end of a basic

S. A. Asghari and H. Pedram is with the Department of Computer Engineering and Information Technology, Amirkabir University of Technology, Tehran, Iran (email: seyyed_asghari@aut.ac.ir, pedram@aut.ac.ir).

H. Taheri is with the Department of Electrical Engineering, Amirkabir University of Technology, Tehran, Iran (email: htaheri@aut.ac.ir,).

O. Kaynak is with the Department of Electrical Engineering, Bogazici University, Istanbul, Turkey (e-mail: okyay.kaynak@boun.edu.tr).

block to any point of another basic block,

Type 4: an error caused by a jump from any point of a basic block to any point of another basic block,

Type 5: an error caused by a jump from any point intra a basic block to a point intra the same block,

Type 6: Errors caused by a jump from any point intra a basic block to a space inter basic blocks (this type is equivalent to illegal jumps from a basic block to the unused space of the memory. In other words, unused space refers to the space inter basic blocks).

It should be noted that in industrial applications, whatever approach is used, whether it is, software or hardware-based, it should be able to handle the errors mentioned above as much as possible and, in doing so, impose as little memory overhead and as little increase in execution time as possible. The latter is commonly referred to as performance overhead in literature and in what follows we will also refer to it as such. We propose a novel fitness factor in this paper that can compare different approaches, based on their fault coverage, memory and performance overheads. It is to be noted that there exists a tradeoff between these parameters, depending on the number of redundant instructions inserted for checking the control flow of the program. However it is important to appreciate that these parameters are all important and the method adopted should consider all of them as much as possible.

The method presented in this paper is effective and applicable in all industrial processes in which a controller is used which may be an embedded system or built on a PLC, a PC, a microcomputer and as such. With the technological developments of recent years, the embedded systems are seen more commonly and in the following parts of the paper the reference will be to such systems.

In the second section of this paper, the related works on hardware and software control flow checking methods are reviewed. The third section introduces the proposed method. The experimental and analytical results of different methods are given in the forth section of the paper.

II. Review of the Literature

Computer systems (especially embedded real time systems) are subject to transient faults due to gamma-rays, x-rays, protons, neutrons and energetic photons. These parameters induce ionization which increases immediate and delayed voltages in devices. These in turn cause transient behaviors in circuits and systems that can disrupt the operation and functionality of the system. The occurrence of transient faults in computer systems during the running of the program resulted in a well-known concern in microelectronic systems since these faults may lead to considerable disruptions and damages. For example, undesired modifications of storage memory cells may occur.

Control flow error detection is one of the effective techniques for achieving reliability. In order to detect such errors, many methods have been proposed since 1980s that can be divided into two categories as hardware and software-based techniques.

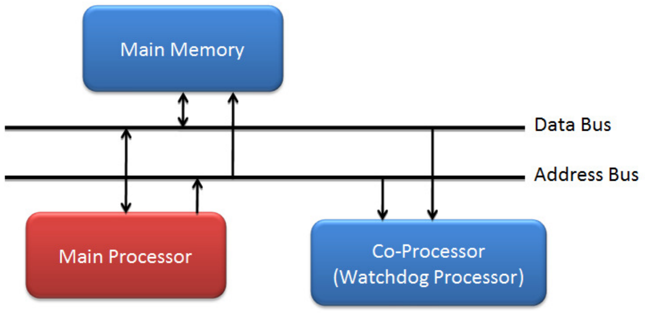


Fig. 1. The structure of a system with watchdog processor

One of the classical hardware methods for control flow checking is the use of a watchdog processor. Watchdog processor is a processing element can detect control flow errors by monitoring the processor behavior on the communication bus. Fig. 1 shows a structure in which a watchdog processor is used for control flow checking [16-19]. In the first phase, information gathered through monitoring the processor and the bus is given to the watchdog and then in the next phase (while the program is running), the watchdog as a co-processor can monitor the flow of the program. Fault detection process is completed when the collected information, such as memory access mechanism, control flow, control signals, and logical results, is compared with the information gathered in the first phase.

In software-based methods, the general procedure of operation is similar to the hardware-based methods. The main difference is that in software methods, the control flow checking is performed by the main processor; instead of any additional hardware [14-17]. The basis of CFC (Control Flow Checking) methods is comparing the control graph of the running program with the one predicted at the beginning of the program. Software based methods are usually performed by code and data (signature) insertion which can be done at the procedure level or the statement level [20-25]. Some machine-level instructions may also be added to the program. Moreover, the running program of the processor is divided into some basic blocks that are branch-free and a signature is assigned to or derived from each block. During the running of the program, control flow is checked by these signatures until the correct points of program blocks are entered and exited. Figure 2 shows the program partitioning into basic blocks and the partitioning instructions. The program is shown by a direct graph in which each node shows an instruction of the machine and the edges are the control flow [14-17]. In Fig. 2(a), the instruction 2 is a branch instruction and so is a divider between basic blocks. The instruction 4 that is the destination of a branch is also used for partitioning between blocks. In this way and as it can be seen in Fig. 2(b), the main program is divided into three basic blocks.

Due to the special environment in space (the existence of high radiation in this environment and its destructive effects on electronic equipment) and the pressing need of high reliability in this environment, numerous works have appeared in literature for enhancing the reliability of space equipment [26-32]. One of the works in this field is Relationship Signature CFC (RSCFC) [20] 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 existing relationships are coded in it. The faults in the control flow of the program are detected by ANDing the runtime signatures with the information at the beginning and end of the blocks. In comparison to the previous works, this method has more fault coverage and a better efficiency and it also consumes less memory [21].

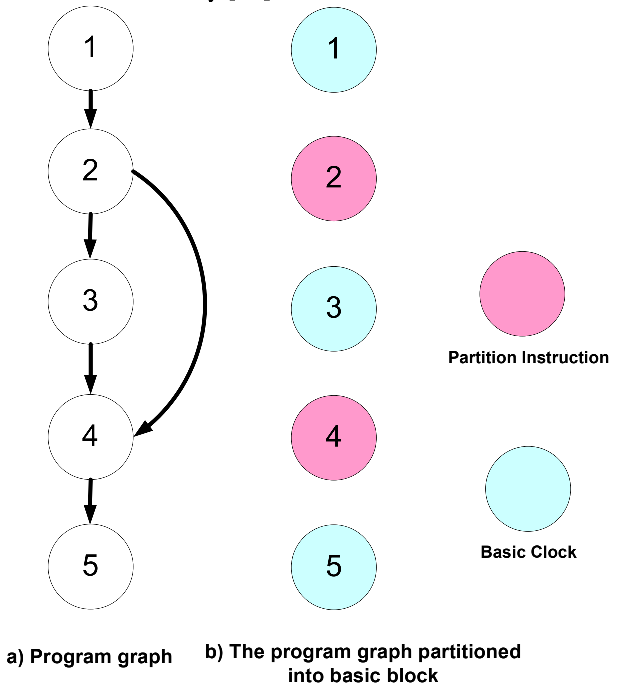


Fig. 2. Program dividing into some basic blocks

In Control Flow Checking by Software Signature (CFCSS) [21] method, a signature of *s* and a signature of *d* are assigned to each basic block. A global variable of *G* is also added to the program which consists of a running block signature amount. During the running of the program, whenever the program enters into a new basic block, *G* is updated to a new amount [17, 21].

The Enhanced Control Flow Checking using Assertion (ECCA) [22], that is another control flow checking method, is implemented in high level of Register Transfer Level (RTL). In high level implementation, ECCA adds a prime number, two instructions and an ID, to each basic block.

III. The Proposed Method

In the previous section, some of the control flow error detection techniques based on software and hardware were reviewed. SCFC method is a software-based one which benefits from the merits of these kinds of techniques. Like other methods in this field, SCFC divides the program into some basic blocks and assigns a signature for each block.

Figure 3 shows a control flow error between two basic blocks that causes an illegal jump from the middle of the block 1 to the block number 2. At the end of the basic block 2, this error is detected and the program control is transferred to the function (Exception Handler or CFE Manager) that can correct this error.

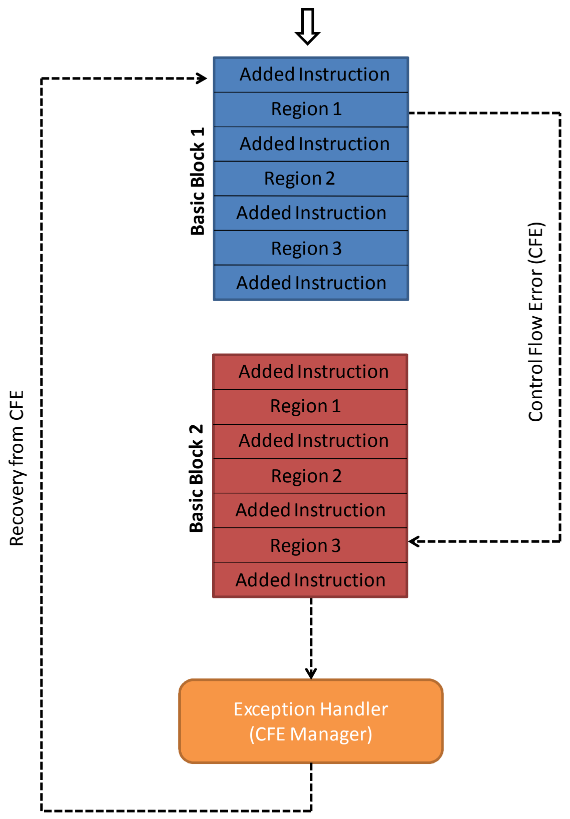


Fig. 3. Detection and correction of control flow errors in processors

SCFC assigns a signature to each basic block like other methods of this field. This signature is S_i variable that shows the successor and the destination blocks of the present block.

SCFC inserts four instructions in each basic block. The first instruction is *control* that checks the entrance flow at the beginning of each basic block. The goal of *control* is to detect illegal jumps to the beginning of basic blocks. The *ID* variable (an identification assigned to each basic block that identifies the order of every basic block running in control flow graph) that is updated at the end of all basic blocks and initialized to zero, is compared with the destination of the block number that is saved in each block and any inconsistency means that the destination of this jump is not the current block or that the *ID* variable is not updated in the last block and the flow is transferred from the intra of the last block to the present block. In both cases, a control flow error is occurred.

The second redundant instruction is called *check* and its task is to confirm that the destination is assigned correctly and the current running block is one of the successors of the source basic block. For monitoring the correctness of the availability, Equation 1 is utilized:

$$\text{error} = S\,[\text{ID}]; \qquad (1)$$

S is S_i variable that is updated during the program running. If ID^{th} bit that shows the present basic block number equals 1 at the middle of each basic block, the destination has been assigned correctly. Otherwise, *error signal*, which is the sign of an error, is activated and the program is stopped. *Check* instruction is inserted in the middle of a basic block to detect control flow error in case of an illegal jump occurrence to the intra of a basic block. In this, illegal jumps from the beginning and intra of the last basic block to the intra of the next basic block can be detected.

The third instruction is called *update* and updates S at runtime. For updating control flow signature, Equation 2 is designed:

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

Therefore variable S is updated in the middle of each basic block in preparation for going to the next destination. It should be noted that S is set to 000…1 for the first time to be able to go to the first basic block. Any other jump is therefore not allowed. This redundant instruction is inserted in the intra of a basic block to detect illegal jumps inter the block. By inserting the present block signature in the signature variable, the next legal successors that should be run, are assigned.

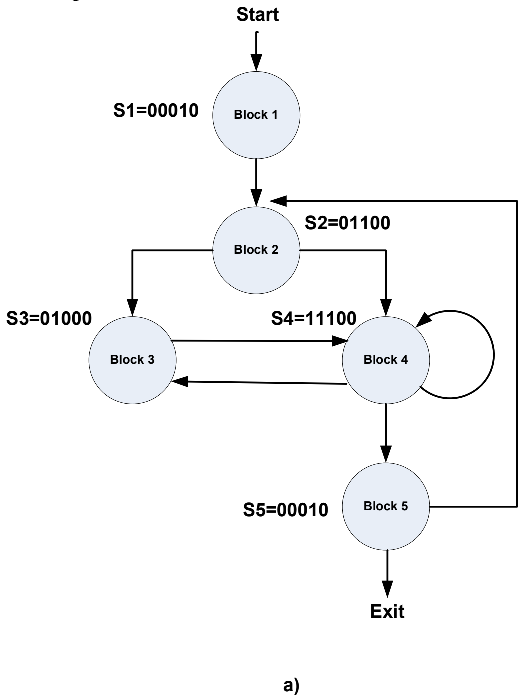
The last instruction is called *exit* that is run at the end of each basic block and updates the *ID* variable to the number that shows the present basic block.

Check and *update* instructions are placed in the middle of each basic block and in this way some of the errors caused by illegal internal jumps in a specific basic block are detected. After the detection an error, *error signal* equals 0 and the program will stop.

It should be mentioned that the assigned signature of each basic block in this method, has N bits. N shows the number of basic blocks in the program control flow graph. Bits related to successor nodes of the present block equal 1 in N bit of the signature. Figure 4 shows a sample program code, control flow graph and the signatures of each block. The sample program considered has 5 basic blocks; therefore, the signature of control flow graph related to each block has 5 nodes and bits. For example, successor nodes of the second basic block of the graph are the blocks 3 and 4 and therefore the signature of this basic block is 01100.

Figure 4(a) shows control flow graph of a sample program and the derived signatures of each block derived signatures. Figure 4(b) shows the structure of a basic block and its redundant instructions. Basic blocks interconnections of a sample program are shown in the control flow graph of Fig. 4(a). As shown in this figure, the signature of each basic block is derived from its successor blocks. Figure 4(b) shows the interior structure of each basic block after inserting the redundant instructions.

Figure 5 shows more basic block structure changes shown in Fig. 4.



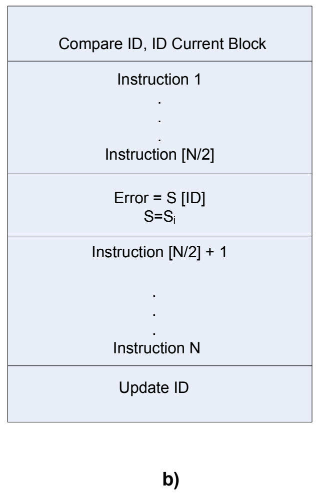


Fig. 4. (a) Control graph of sample bubble sort program, (b) the interior structure of a basic block by inserting redundant instructions

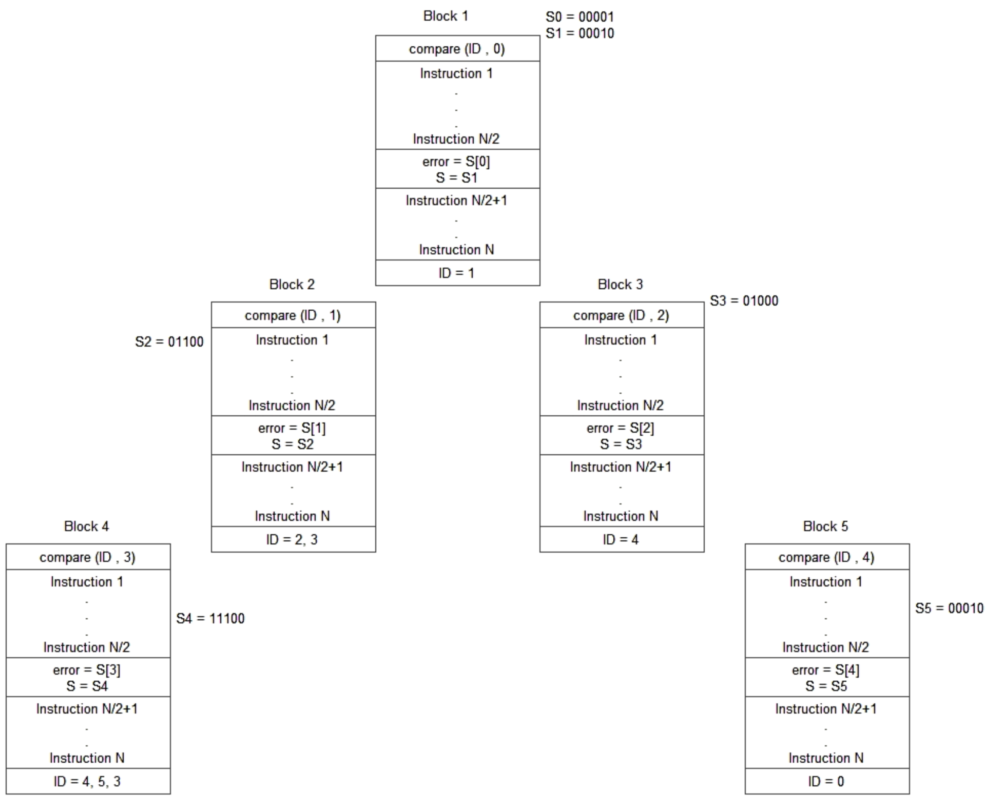


Fig. 5. Basic block structure changes

All of the single errors caused by incorrect jumps can be found by inserting redundant instructions. The proof of this claim is explained in the following:

A. *Illegal jump from node vi to node vj*

When an illegal jump occurs from block *vi* to the block *vj*. In these cases, when check instruction is run in *vj* block, since *ID* variable is not updated in block vj, *error* signal becomes zero and this error is detected. For example, imagine in Fig. 4, there is an unwanted jump from the end of the first basic block to the middle of the second block. In this case, since *ID* variable at the end of the first block is not updated, its amount remains zero. When operation is reached to *check* and *update* instructions of the second basic block, the first bit (zero bit position) of variable S, that now has *Signature of Basic Block 1* (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 of Basic Block 1* = 00010,

S [zero bit position] = 0,

error = S [zero bit position] = 0

B. *Illegal jump from node vi to itself*

When an illegal jump occurs from one instruction before *check* and *updates* to the instructions after them. In this case, since S has not been updated during program running, error is detected in check instruction of the next block. For example, imagine an unwanted jump occurs from the beginning of the first block to the middle or end of the same block. In this case, the *update* instruction has not been run so *S* takes its initial amount that is 00001. At the end of the first block, *ID* amount is updated to 1 and the program enters the second basic block. In the second basic block and in check instruction, the first bit of S is updated to 1 and the occurred error will be detected:

$$S = S_{initial} = 00001, \text{err} = S\,[ID] = 0$$

IV. Experimental Results

In this section, the environment used for tests is described and the experimental results are given.

A. *Test environment*

For analyzing the proposed method, the infrastructure shown in Fig. 6 is utilized which contains the following elements as main parts [33]:

- Background Debug Mode (BDM) module. This component is a programming tool that can be used for debugging and fault injection. It is a tool which Motorola Corporation placed it in their microprocessors and microcontrollers.
- PhyCORE-MPC555 (a product of a PHYTECH technology holding company) evaluation board
- A personal computer

An additional technique that is used for fault injection is the manual manipulation of the jumps of the program as follows:

- Direct fault injection into processor registers by the use of BDM module in bit flip model (the conversion of 0 to 1 and vice versa);
- Applying jump instructions to the program (JMP, JL, JG, JNE, JLE, JGE, CALL and RET);
- Changing jump instructions.

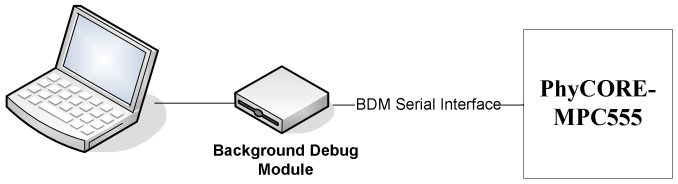


Fig. 6. Fault injection mechanism structure by the use of BDM [33].

Fault injection operation is applied to three benchmark programs, Bubble Sort (BS), Quick Sort (QS) and 40×40 Matrixes Multiplication (MM) and about 5000 faults are injected to 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 that is close to reality. For example, in this method, PC (Program Counter) register and SR (Status Register) can be directly manipulated. On the other hand as it is shown in [33], exception occurrence probability is very high by manipulating registers. Therefore, besides this method another fault injection approach is also used that has three kinds:

- Random jump deletion in the program, in which some branches of the program are deleted randomly;
- Random jump changing, in which some branches of the program or their operands changes randomly;
- Random jump insertion, in which some branches are inserted at different parts of the program.

By the mentioned fault injection methods, the control flow errors will be produced and the efficiency of different methods can be compared with each other. The faults are injected to the assembly code of benchmarks and at random places of it. On the other hand, by using BDM method, registers and program counter of the program are changed and control flow errors occur in the program. Therefore, the efficiency of different methods can be evaluated.

Five versions are considered for each benchmark for fault injection:

- The original code (the code of the benchmark);
- Adding CFCSS method to the original code;
- Adding ECSS method to the original code;
- Adding RSCFC method to the original code;
- Adding SCFC method to the original code.

For each version, the program is compiled and its assembly code is generated. Then one method of fault injection is randomly selected that changes the program or its registers. Finally the faulty code is compiled and executed. This process is repeated 5000 times for each of the mentioned versions.

The injected faults can result in five different cases according to the effect they produce in the running of the program:

- **CR (Correct Result):** the fault does not change the final result of the program
- **OS (Operating System):** the fault is detected by operating system and its exceptions
- **WR (Wrong Result):** the fault changes the final result of the program and produces a wrong output
- **TO (Time Out):** the fault changes the 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 occurrence percentages of these cases are shown in Table 1. It should be noted that the fault coverage of each method is equal to its Single Detection (SD) percentage since the first four cases can be detected without the use of any control flow error detection method. In other words, Single Detection (SD) percentage that is shown in the last column of Table 1 indicates the error detection capability of the proposed method embedded into the stated benchmark. Since three numbers are given for 3 different benchmarks, we can derive an overall number that describes the single detection capability of the methods considered by taking the averages of these 3 numbers. For example for the proposed SCFC method the average is:

$$43.55\% + 48.56\% + 49.70\ \% = 47.27\ \%$$

For the other methods, the corresponding figures are 31.81%, 33.84% and 32.10% respectively. Therefore, single detection capability is increased by 14.7% on average in the SCFC method, in comparison with the other methods.

Figures 7, 8 and 9 compare the fault coverage, the memory and the performance overhead of CFCSS, ECCA and RSCFC methods with the proposed technique of this paper, the word average indicating the average of the figures when the methods are embedded into three benchmarks; BS, MM and QS.

SCFC memory and performance overheads are calculated as follows:

$$\frac{\mathit{Memory\ Storage\ or\ Execution Time\ of\ \ Hardened\ Method}}{\mathit{Memory\ Storage\ or\ Execution Time\ of\ \ Normal\ Method}}$$

In this equation, *Hardened Method* refers to the program running by the use of the proposed method. In this running, software redundancy caused by adding signatures to the basic blocks is considered as overhead (for example, in the proposed method, after dividing the program into basic blocks and adding unique signatures to the basic blocks, the program size is 1.43 times larger as compared to the other methods). *Normal Method* refers to running the program without considering any error detection mechanism. In other words, the proposed method is not utilized in this running.

For performance overhead, this action is repeated with the execution time factor. As it can be seen in Fig. 7, SCFC method in comparison with ECCA, RSCFC, and CFCSS methods increases the fault coverage in average to 6.12%. Meanwhile it has less memory overhead and its performance overhead can compete with other methods (Fig. 8 and 9). As it was mentioned in the previous sections, fault coverage is as important as memory and performance overheads of a system.

An appropriate method should be able to balance these parameters with each other and does not consider only one of them. For this purpose, a new parameter, called Evaluation Factor (EF), is introduced in this paper. It considers overheads and fault coverage of each method as shown in Equation 3:

$$Evaluation\,Factor = \frac{Fault\,Coverage}{Memory\,Overhead \times Performance\,Overhead} \quad (3)$$

The averages of the evaluation factors of the different methods when embedded into the 3 benchmark programs are shown in Table 2. As it can be predicted, SCFC is better than previously mentioned methods considering fault coverage, performance and memory overheads.

As shown in figures and tables and based on the experiments, the proposed method of this paper outperforms others in fault coverage and overheads. Due to the ability of SCFC in intra block control flow error checking, it can detect more errors than other techniques. It also takes less instructions and variables than other methods for control flow error detection. In the following, the fault coverage of different methods is studied analytically.

TABLE 1: FAULT INJECTION RESULTS IN ORIGINAL PROGRAM, CFCSS, ECCA, RSCFC AND SCFC METHODS (CR: CORRECT RESULT, OS: OPERATING SYSTEM, WR: WRONG RESULT, TO: TIME OUT, AND SD: SINGLE DETECTION)

Benchmarks	CR	OS	WR	TO	SD
BS	19.54%	35.68%	37.33%	7.45%	0%
MM	11.76%	38.84%	39.82%	5.58%	0%
QS	20.29%	37.65%	36.72%	5.34%	0%
BS-CFCSS	45.65%	6.88%	11.80%	2.87%	32.80%
MM-CFCSS	44.50%	12.87%	13.14%	3.06%	26.34%
QS-CFCSS	38.13%	11.20%	9.40%	4.96%	36.31%
BS-ECCA	38.05%	13.00%	10.25%	4.10%	34.60%
MM-ECCA	42.70%	11.09%	11.20%	5.87%	29.14%
QS –ECCA	40.70%	6.66%	10.54%	4.30%	37.80%
BS-RSCFC	42.30%	11.10%	11.6%	2.50%	32.50%
MM-RSCFC	40.96%	12.34%	9.7%	2.50%	34.50%
QS –RSCFC	39.09%	13.65%	11.2%	6.56%	29.50%
BS-SCFC	42.23%	6.90%	3.33%	3.99%	43.55%
MM- SCFC	37.20%	6.70%	5.20%	2.34%	48.56%
QS – SCFC	36.20%	6.20%	4.60%	3.30%	49.70%

TABLE 2: AVERAGE VALUE OF EVALUATION FACTORS FOR DIFFERENT METHODS

Techniques	Averages of Evaluation Factors
CFCSS	39.09
ECCA	44.32
RSCFC	35.47
SCFC	47.76

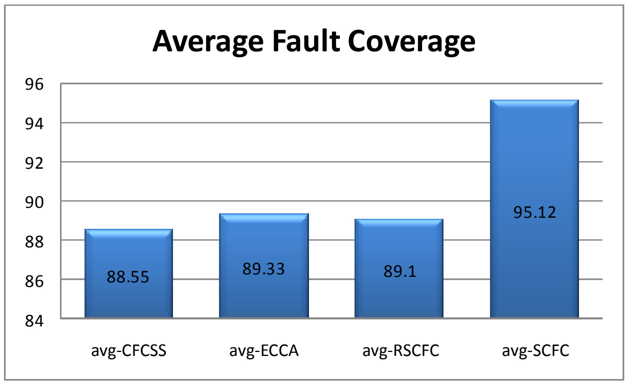


Fig. 7: Total fault coverage comparison

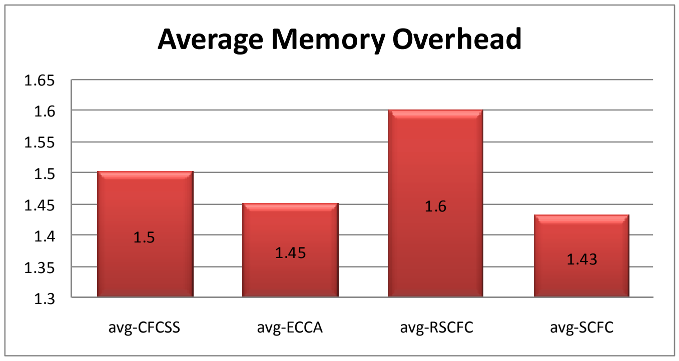


Fig. 8: Memory overhead comparison

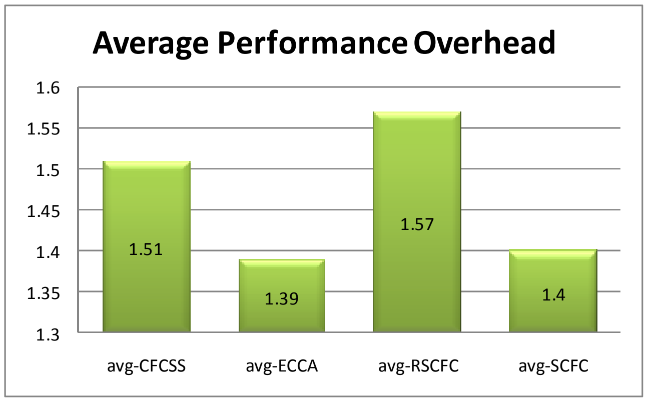


Fig. 9: Performance overhead comparison

B. Analytical computation of fault coverage

In above the effectiveness of the proposed method, is demonstrated by experimental results. In this section, an analytical computation for the fault coverage of the proposed method is presented. The equations show that how much the probability of illegal jumps occurrence that is known as control flow error is by considering the number of instructions and basic blocks. Moreover, by considering the capability of the presented techniques in detecting different control flow errors, it is assigned that which of these probabilities is detected by the presented technique. By considering the number of instructions and the basic blocks, equations are derived that indicate the probability of the occurrence of illegal jumps, i.e. the control flow error.

The following states are considered for analyzing impermissible jumps that lead to control flow errors:

1. **The illegal jump from one basic block to another:** In

this case, the error that occurs in PC register leads to an unwanted jump from a basic block to another incorrect basic block. Therefore, the probability of a case when program flow is in ith basic block and due to an error occurrence is transferred to jth basic block, should be calculated. This probability is calculated in Equation 4:

$$P_{\text{type-1}} = P_{\text{Basic Block}i} * P_{\text{Basic Block } i \longrightarrow \text{Basic Block } j} * (1 - P_{\text{program crash}}) \tag{4}$$

2. **An illegal jump from one basic block to the partition block:** In this case, the error that occurs in PC register leads to an illegal jump from a basic block to a memory space out of the basic blocks. These places are called Partition Block (PB). Therefore the probability of a case in which the program flow is in ith basic block and is transferred to PB basic block due to an error occurrence, should be calculated. This calculation is shown in Equation 5:

$$P_{\text{type-2}} = P_{\text{Basic Block } i} * P_{\text{Basic Block } i \longrightarrow \text{Partition Block}} * (1 - P_{\text{program crash}}) \tag{5}$$

3. **The jump from a basic to itself:** In this case, the error that occurs in PC register leads to an illegal jump from a basic block to a place inside the same basic block. Therefore, the probability of the case in which the program flow is in ith basic block and is transferred to another place in the same ith basic block due an error occurrence should be calculated. This probability is shown in Equation 6:

$$P_{\text{type-3}} = P_{\text{Basic Block } i} * P_{\text{Basic Block } i \longrightarrow \text{Basic Block } i} * (1 - P_{\text{program crash}}) \tag{6}$$

If the error of the third type (the jump from a basic block to the same basic block) is analyzed in more detail, two states can be extracted for illegal jump inside a basic block. One state is when jumps occur from the upper part to the lower part of the basic block and vice versa. Another state is when jumps occur from the upper part to the upper part and from the lower part to the lower part. For calculating the probability of the first state of the third type error, Equation 7 and for the second state, Equation 8 can be used:

$$P_{\text{type-3-1}} = P_{\text{Basic Block } i} * (P_{\text{Basic Block } iu \longrightarrow \text{Basic Block } iD} + P_{\text{Basic Block } iD \longrightarrow \text{Basic Block } iu}) * (1 - P_{\text{program crash}}) \tag{7}$$

$$P_{\text{type-3-2}} = P_{\text{Basic Block } i} * (P_{\text{Basic Block } iu \longrightarrow \text{Basic Block } iu} + P_{\text{Basic Block } id \longrightarrow \text{Basic Block } id}) * (1 - P_{\text{program crash}}) \tag{8}$$

Adding all of the above stated states in one statement, Equation 9 is obtained:

$$P_{\text{Basic Block } i \longrightarrow \text{Basic Block } i} = P_{\text{Basic Block } iu \longrightarrow \text{Basic Block } iu} + P_{\text{Basic Block } id \longrightarrow \text{Basic Block } id} + P_{\text{Basic Block } iu \longrightarrow \text{Basic Block } id} + P_{\text{Basic Block } id \longrightarrow \text{Basic Block} iu} \tag{9}$$

Considering a Bernoulli random variable z that shows the jump direction, the following can be written

$P_{\text{Basic Block } iu \longrightarrow \text{Basic Block } iu}(x, y) = P(P_{\text{Basic Block } iu \longrightarrow \text{Basic Block } iu} \mid z=1)\ P(z=1) + P(P_{\text{Basic Block } iu \longrightarrow \text{Basic Block } iu} \mid z=0)\ P(z=0)$

$P(z = 0) = P(z=1) = 1/2$

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

$P(P_{\text{Basic Block } iu \longrightarrow \text{Basic Block } iu} \mid z=1) = P(L(x) <= y)$

$P(P_{\text{Basic Block } iu \longrightarrow \text{Basic Block } iu} \mid z=0) = P(L(x) <= S_{\text{Basic Block}}/2 - y)$

In above, S_{BB} variable shows the average size of a basic block based on bytes 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 indicates the distance (in bytes) of the place where the fault occurs from the beginning of the basic block [2].

SEU errors in the program counter lead to control flow errors. Consider a random variable *x* between 0 to 31 for the erroneous bit number in the program counter and define *L(x)* with the value 2^x as the variable that indicates the number of impermissible jumps by and changing x^{th} bit in the program counter [2].

For computing jump probability of each basic block to another basic block, Equation 10 is utilized (in this equation, I_{AN} is the average number of instruction bytes):

$$P_{\text{Basic Block } i \longrightarrow \text{Basic Block } j}(x) = 1/N_{\text{Basic Block}} * 1/2 * 1/N_{\text{Partition Block}} \sum_{k=0}^{N_{\text{Basic Block}}} \sum_{i=1}^{N_{\text{Partition Block}}} \sum_{j=1}^{i-1} P(I_{AN}.k + j.S_{\text{Partition Block}} + (j-1)S_{\text{Basic Block}} < L(x) < I_{AN}.k + j.S_{\text{Partition Block}} + j.S_{\text{Basic Block}}) + 1/N_{\text{Basic Block}} * 1/2 * 1/N_{\text{Partition Block}} \sum_{k=0}^{N_{\text{Basic Block}}} \sum_{i=1}^{N_{\text{Partition Bloack}}} \sum_{j=1}^{N_{\text{Partition Block}}-i} P(S_{\text{Basic Block}} - I_{AN}.k + j.S_{\text{Partition Block}} + (j-1)\ S_{\text{Basic Block}} < L(x) < S_{\text{Basic Block}} - I_{AN}.k + j.S_{\text{Partition Block}} + j.\ S_{\text{Basic Block}}) \tag{10}$$

The jump probability from one basic block to a place outside the program is computed by Equation 11:

$$P_{\text{Basic Block } i \longrightarrow \text{Basic Block } j}(x) = 1/N_{\text{Basic Block}} * 1/2 * 1/N_{\text{Partition Block}} \sum_{k=0}^{N_{\text{BasicBlock}}} \sum_{i=1}^{N_{\text{Partition Block}}} \sum_{j=1}^{i-1} P(I_{AN}.k + (j-1).S_{\text{Partition Block}} + (j-1)S_{\text{Basic Block}} < L(x) < I_{AN}.k + j.S_{\text{Partition Block}} + (j-1).S_{\text{Basic Block}}) + 1/N_{\text{Basic Block}} * 1/2 * 1/N_{\text{Partition Block}} \sum_{k=0}^{N_{\text{Basic Block}}} \sum_{i=1}^{N_{\text{Partition Block}}} \sum_{j=1}^{N_{\text{Partition Block}}-i} P(S_{\text{Basic Block}} - I_{AN}.k + (j-1).S_{\text{Pasic Block}} + (j-1)\ S_{\text{Basic Block}} < L(x) < S_{\text{Basic Block}} - I_{AN}.k + j.S_{\text{Partition Block}} + (j-1).\ S_{\text{Basic Block}}) \tag{11}$$

It is seen that SCFC method should have a better fault coverage in comparison with the other methods due to being able to detect a percentage of the first kind of Type 3 jump (Type 3-1). Table 3 shows the results of applying SCFC method on Quick Sort, Matrix Multiplying, and Bubble Sort benchmarks in terms of the detection percentage of each kind of jump.

TABLE 3: RESULTS OF APPLYING SCFC METHOD ON THREE BENCHMARKS

Benchmarks	P_{type-1} (%)	P_{type-2} (%)	P_{type-3}* (%)
QS	74.7	15.7	9.6
MM	72.1	16.7	11.2
BS	78.6	12.6	8.8
Average	75.13	15	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, $P_{type\ 3\text{-}2(Average)}$=5.53

It can be concluded from the equations given and the figures of Table 3, SCFC is able to detect Type1, Type 2 and Type 3-1 (as shown in the bottom part of Table 3) errors. As it was mentioned before, other error detection methods do not have Type 3-1 error detection capability. Therefore, it can be derived from Table 3 figures that SCFC method has at least 4.33% more error detection capability than other methods. This figure is consistent with the results of experimental methods.

V. CONCLUSION

This paper presents a new control flow checking method, SCFC, which divides the program into some basic blocks and inserts redundant instructions and a signature in them. The bit number of the signature is equal to the number of basic blocks and is derived from the control flow graph of the program and based on the successors of each block. SCFC inserts some redundant instructions in the middle of basic blocks, so that it can detect a percentage of illegal intra block jumps beside inter block ones. On the other hand, this method has less memory and performance overheads in comparison with the other proposed methods in this field. Errors that occur in harsh industrial environments, such as in space environments, may lead to destructions that can have very costly results such as human hazards and the loss of very costly equipment. Therefore, increasing reliability in these systems is very important. The method proposed in this paper therefore carries a greater importance in comparison with similar works in this field.

In this paper, a new metric called Evaluation Parameter is introduced that simultaneously considers fault coverage, memory and performance overheads. In this way each method can be evaluated efficiently. SCFC is used with three standard benchmark programs of this field, i.e., bubble sort, quick sort and matrix multiplication. The fault coverage is computed experimentally and analytically. The results obtained indicate that SCFC has better fault coverage and less memory and performance overheads in comparison with the previously proposed methods in literature. One other difference of the proposed method from the other software-based methods is the utilization of a hardware fault injection tool that decreases the fault injection time. Furthermore, the injection environment is, in this case, more similar to the real environment. For example, SR (Status Register) and PC registers can be directly manipulated.

ACKNOWLEDGMENT

We would like to acknowledge Ms. Atena Abdi who helped us in simulations.

REFERENCES

[1] A. Rajabzadeh, G. Miremadi, and M. Mohandespour, "Error detection enhancement in COTS superscalar processors with performance monitoring features," *J Electron Testing: Theory Appl (JETTA)*, vol 20, pp. 553–67, 2004.

[2] A. Rajabzadeh, and G. Miremadi, "Transient detection in COTS processors using software approach," *Elsevier Journal of Microelectronics Reliability*, vol 46, pp. 124-133, January 2006.

[3] J. Srinivasan, and K. Lundqvist, "Real-time architecture analysis: a COTS perspective," Presented at the 2002 in *Proc. 21th digital avionics systems*, pp. 5D4-1–9.

[4] Y. He, and A. Avizienis, "Assessment of the applicability of COTS microprocessors in high-confidence computing systems: a case study," Presented at the 2000 *Int. Conf. Dependable Systems and Networks (DSN- 2000)*, pp. 81–6.

[5] CD. Gill, RK. Cytron, and DC. Schmidt, "Multiparadigm scheduling for distributed real-time embedded computing," *IEEE Journal*, vol 91, pp. 183–97, January 2003.

[6] N. Oh, PP. Shirvani, and EJ. McCluskey, "Error detection by duplicated instructions in super-scalar processors," *IEEE Trans. Reliab*, vol 51, pp. 63–75, 2002.

[7] A. Malinowski, and H. Yu, "Comparison of Embedded System Design for Industrial Applications," *IEEE Trans. Industrial Informatics*, vol. 7, pp. 244-254, May 2011.

[8] Y. Zhang, H. Zhou, S. J. Qin, and T. Chai, "Decentralized Fault Diagnosis of Large-Scale Processes Using Multiblock Kernel Partial Least Squares," *IEEE Trans. Industrial Informatics*, vol. 6, pp. 3-10, February 2010.

[9] M-D Ma, D.S.-H. Wong, S-S Jang, and S-T Tseng, "Fault Detection Based on Statistical Multivariate Analysis and Microarray Visualization," *IEEE Trans. Industrial Informatics*, vol. 6, pp: 18-24, February 2010.

[10] M. H. Kim, S. Lee, and K. C. Lee, "Kalman Predictive Redundancy System for Fault Tolerance of Safety-Critical Systems," *IEEE Trans. Industrial Informatics*, vol. 6, pp. 46-53, February 2010.

[11] P. Conmy and I. Bate, "Component-Based Safety Analysis of FPGAs," *IEEE Trans. Industrial Informatics*, vol. 6, pp. 195-205, May 2010.

[12] G. Gaderer, P. Loschmidt, and T. Sauter, "Improving Fault Tolerance in High-Precision Clock Synchronization," *IEEE Trans. Industrial Informatics*, vol. 6, pp: 206-215, May 2010.

[13] A. Quagli, D. Fontanelli, L. Greco, and L. Palopoli, A. Bicchi, "Design of Embedded Controllers Based on Anytime Computing," *IEEE Trans. Industrial Informatics*, vol. 6, pp. 492-502, November 2010.

[14] D. Zhu, and H. Aydin, "Reliability Effects of Process and Thread Redundancy on Chip Multiprocessors," Presented at the 2006 in *Proc. of the 36th Annual IEEE/IFIP International Conference on Dependable Systems and Networks*.

[15] M. Jafari-Nodoushan, G. Miremadi, and A. Ejlali, "Control-Flow Checking Using Branch Instructions," Presented at the 2008 in *Proc. of the 8th Int. Conf. on Embedded and Ubiquitous Computing*.

[16] S. A. Asghari, A. Abdi, H. Taheri, S. Pourmozaffari and H. Pedram, "SEDSR: Soft Error Detection using Software Redundancy," *Journal of Software Engineering and Applications,* vol. 5, pp. 664-670, 2012.

[17] S. A. Asghari, A. Abdi, H. Taheri, H. Pedram, S. Pourmozaffari, "I2BCFC: An Effective Intra-Inter Block Control Flow Checking Method against Single Event Upsets," *Research Journal of Applied Sciences, Engineering and Technology*, vol. 4, pp. 4367-4379, 2012.

[18] P. Kongetira and K. Aingaran, and K. Olukotun, "Niagara: A 32-Way Multithreaded Sparc Processor," *IEEE Micro*, vol 25, pp. 21–29, 2005.

[19] 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, pp. 959-972, May 2006.

[20] A. Li, and B. Hong, "On-line control flow error detection using relationship signatures among basic blocks," *Computers and Electrical Engineering Journal*, Elsevier, vol 36, pp. 132–141, 2010.

[21] N. Oh, P. P. Shirvani, and E. J. McClusky, "Control Flow Checking By Software Signature," *IEEE Trans. Reliab*, vol 5, pp. 111-122, 2002.

[22] 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 Trans. Parallel Distributed Systems*, vol 10, pp. 627–641, 1999.

[23] L. Jianli, T. Qingping and Xu. Jianjun, "A Software-Implemented Configurable Control Flow Checking Method," Presented at the 2010 in

Proc Int Symposium on Parallel Architectures, Algorithms and Programming (PAAP).

[24] M. J. Gadlage, 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, "Assessing Alpha Particle-Induced Single Event Transient Vulnerability in a 90-nm CMOS Technology," *IEEE Electron Device Letter*, vol 29, pp. 638-640, 2008.

[25] J. A. Felix, J. R. Schwank, M. R. Shaneyfelt, J. Baggio, P. Paillet, V. Ferlet-Cavrois, P. E. Dodd, S. Girard, and E. W. Blackmore, "Test Procedures for Proton-Induced Single Event Latchup in Space Environments," *IEEE Trans. Nuclear Science*, vol 55, pp. 2161-2165, August 2008.

[26] K. Kruckmeyer, R. L. Rennie, and V. Ramachandran, "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. Nuclear Science*, vol 55, pp. 113-117, August 2008.

[27] M. R. Shaneyfelt, J. R. Schwank, P. E. Dodd, and J. A. Felix, "Total Ionizing Dose and Single Event Effects Hardness Assurance Qualification Issues for Microelectronics," *IEEE Trans. Nuclear Science*, vol 55, pp. 1926-1946, August 2008.

[28] J. A. Maestro, and P. Reviriego, "Reliability of Single-Error Correction Protected Memories," *IEEE Trans. Reliab*, vol 58, pp: 193-201, March 2009.

[29] E. L. Petersen, "Single-Event Data Analysis," *IEEE Trans. Nuclear Science*, vol 55, pp. 2819-2841, December 2008.

[30] S. Askari and M. Nourani, "Design methodology for mitigating transient errors in analogue and mixed-signal circuits," *Circuits, Devices & Systems, IET*, vol.6, no.6, pp.447-456, Nov. 2012.

[31] H.R. Mahdiani, S.M. Fakhraie and C. Lucas, "Relaxed Fault-Tolerant Hardware Implementation of Neural Networks in the Presence of Multiple Transient Errors," *IEEE Trans. Neural Networks and Learning Systems*, vol.23, no.8, pp.1215-1228, August 2012

[32] C. Hyungmin, L. Larkhoon and S. Mitra, "ERSA: Error Resilient System Architecture for Probabilistic Applications," *IEEE Trans. Computer-Aided Design of Integrated Circuits and Systems*, vol.31, no.4, pp.546-558, April 2012.

[33] 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 (IREMOS)*, vol. 3, pp. 415-422, 2010.

Seyyed Amir Asghari was born in Lashte Nesha in Guilan province of Iran, on June 26, 1984. He received his BS degree from Amirkabir University of Technology in 2007 and MS degree from the same university in 2009 in Computer Engineering. He is a PhD Candidate currently and his interests include reliable and fault tolerant embedded system design, real-time system design and operating systems.

Hassan Taheri Received his BS degree from Amirkabir University of Technology in 1975 and MS degree from University of Manchester Institute of Science and Technology (UMIST) in 1978 in Electrical Engineering. He received his PhD degree from UMIST University in 1988 in Electrical Engineering. Dr. Taheri 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.

Hossein Pedram Received his BS degree from Sharif University in 1977 and MS degree from ohio State University in 1980 in Electrical Engineering. He received his PhD degree from Washington State University in 1992 in Computer Engineering. Dr. Pedram has served as a faculty member in the Computer Engineering Department in Amirkabir University of Technology since 1992. He teaches courses in computer architecture and distributed systems. His research interests include innovative methods in computer architecture such as asynchronous circuits, management of computer networks, distributed systems, and robotics.

Okyay Kaynak (M'80-SM'90-F'03) received the B.Sc. (first-class honors) and Ph.D. degrees in electronic and electrical engineering from the University of Birmingham, Birmingham, U.K., in 1969 and 1972, respectively. From 1972 to 1979, he held various positions in the industry. In 1979, he Joined the Department of Electrical and Electronics Engineering, Bogazici University, Istanbul, Turkey, where he is currently a Full Professor, holding the UNESCO Chair on Mechatronics. He has held long-term (near to or more than a year) Visiting Professor/Scholar positions with various institutions in Japan, Germany, the U.S., and Singapore. His current research interests include intelligent control and mechatronics. He is the author of three and the editor of five books. In addition, he is the author or coauthor of close to 350 papers that have appeared in various journals and conference proceedings. Dr. Kaynak is active in international organizations, has served on many committees of the IEEE, and was the President of the IEEE Industrial Electronics Society during 2002–2003. Currently he is on the IEEE MGA Board.