Masking Wrong-successor Control Flow Errors Employing Data Redundancy

Javad Yousefi[1], Yasser Sedaghat[2], Mohammadreza Rezaee[3]
Dependable Distributed Embedded Systems (DDEmS) Laboratory
Department of Computer Engineering, Ferdowsi University of Mashhad
Mashhad, Iran
[1]javad.yousefi@stu.um.ac.ir, [2]y_sedaghat@um.ac.ir, [3]mohammadreza.rezaee@stu.um.ac.ir

Abstract— **Advancements of CMOS technology lead to reduction of the transistor size and operating voltage levels that cause transistors to become more sensitive to cosmic rays. Therefore CMOS devices like memory (i.e., RAMs) are more likely to be hit by transient faults. Up to 77% of the transient faults cause Control Flow Errors (CFEs). One type of CFEs is wrong-successor CFE which is caused by faults in data variables resident in RAM. Previous control flow checking techniques neither detect nor correct this type of errors. A technique with the ability of masking wrong-successor CFEs is proposed in this paper. Since occurrence of these errors is induced by faults in data variables which affect the program execution flow (control variables), in the proposed technique, the control variables are being distinguished from other variables. This step is being followed by a traditional fault masking technique that is applied on the control variables. To evaluate the proposed technique, it was applied on five various benchmarks of the MiBench package. The experimental results demonstrated that the proposed technique is able to mask all 50,000 injected faults in control variables; while it had almost 21% performance overhead with 6% memory overhead. It is reasonable and feasible to apply this technique on the former control flow checking techniques due to its perfect wrong-successor CFE correction coverage and low overheads.**

Keywords— Wrong-successor control flow error, Error correction, Control variables, fault masking.

I. INTRODUCTION

Safety-critical systems are the systems whose a failure may result in catastrophic consequences such as loss or serious hurt to people, extreme harm to property and environmental scathe. Medical devices along with aircraft flight control systems and railway signaling systems are some famous examples of safety-critical systems that have a remarkable impact on human's daily life. These systems usually operate in harsh environments; therefore, fault tolerance is the key requirement of these systems [1].

As the CMOS technology is counting to reduce dimensions and operating voltage levels of computer electronics, the radiation sensitivity of computer electronics such as memories extensively increases; therefore, safety-critical systems are much more fault prone [2].

Categorizing faults by their time span produce three main types that may threat the computer systems. These types of faults are permanent, intermittent and transient faults. Transient faults or soft errors have the most occurrence among all by the probability 10 to 30 times greater than the other types [3].

A transient fault is a fault that its effect fades away having time passed. Therefore, a component affected with a transient fault is able to function normally after spending this time [4]. Transient faults that have the origin of alpha particles from decaying radioactive impurities in packaging and interconnect materials, and high-energy neutrons from cosmic radiation are one of the most important threats in memory devices [5, 6]. These faults can cause a bit-flip in the memory cells. A bit-flip is an unwanted change in the state of a memory cell. It can change the state of the memory cell from 0 to 1 or from 1 to 0 [7]. This change can modify an instruction of the program or alter the stored data in the memory. This modifications may cause a Control Flow Error (CFE) or data error in the program. A CFE occurs when the processor executes a different sequence of instructions compared to the desired execution. When the value of a variable in the program alters erroneously, a data error would arise [7, 8]. For example, a fault in the code segment of the program may cause a change in the opcode of a non-control instruction. This modification may alter the instruction to a control instruction, so this fault resulted as a CFE. If the same fault exists in the data segment of the program, this fault may change the value of a variable in the program that causes a data error [9].

It is reported that up to 77% of the transient faults cause CFEs. Some type of the CFEs occur due to faults in data. For example, if a fault occurs in a variable that is stored in the memory and a branch decision make up its mind by the value of the faulty variable then the fault may cause a CFE. This type of CFE is called a wrong-successor CFE. In fact, a valid yet incorrect branch is happening in this scenario. In the case that an invalid branch happens this branch is called not-successor [10, 11].

Control flow checking techniques are widely being applied to detect CFEs since the 1980s. These techniques are categorized into three major groups, software-based,

978-1-4673-9280-8/15/$31.00 ©2015 IEEE

hardware-based and hybrid [11, 12]. The signature monitoring method is the foundation of the most of these control flow checking techniques. In this method the program is divided into basic blocks. A basic block is a maximal set of ordered instructions in which its execution begins from the first instruction and terminates at the last instruction. There is no branching instruction in a basic block except possibly for the last one. A graph can be produced by the control flow of the program which is known as the Control Flow Graph (CFG). Basic blocks constitute the vertices of the graph while edges represent the branches [13]. Fig.1 shows the basic blocks and an example of how the CFG of the insertion sort procedure is generated.

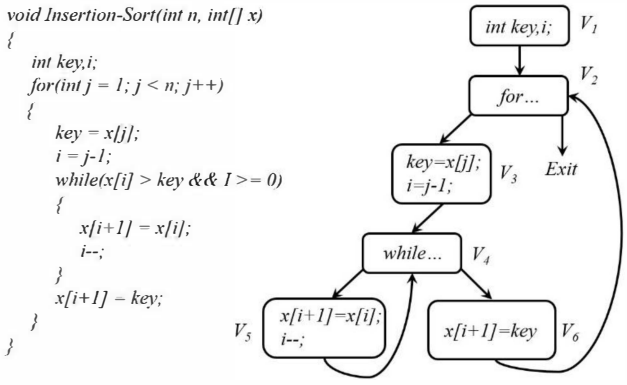

Figure 1. Basic blocks and CFG of insertion sort procedure

In signature monitoring methods, at the compile time, a signature is being dedicated to any vertex of the graph, after having the control flow graph made. At the execution time of the program, a signature is made that is saved based on the control flow checking technique. This signature always indicate a basic block of code which is being executed by the system. When the control of program leave a basic block this signature, will be compared with the dedicated signature then it will be updated with the signature of the destination block. If the mentioned signatures do not be the same, a control flow error had happened and these techniques detect this error. In hardware-based control flow checking techniques, creating the run-time signature and comparing it with the dedicated signature are the duty of a redundant hardware like a watchdog processor. yet, in the software-based control flow checking techniques, this responsibility is done only using software by adding some extra codes to the program in order to create and compare the signatures [11, 13].

Although former control flow checking techniques have a great rate of control flow error detection, in evaluation of these techniques some form of targeted fault injection has been used and there is no fault injection in data; therefore, wrong-successor CFEs were not considered entirely. The capability of these techniques to detect wrong-successor errors were not completely evaluated. Moreover, since the control flow checking techniques focus on the control flow of the programs they are only able to detect not-successor control flow errors not the wrong-successor ones [10].

This paper focuses on masking wrong-successor CFEs which are caused by faults in data variables. These data variables, which are called control variables, affect the execution flow of the program. The idea behind this paper has two main phases. In the first phase, control variables of the program are being differentiated and in the second phase, a traditional fault masking technique is applied on the control variables. Implementations of this idea show that it is possible to mask all the wrong-successor CFEs with a slight performance and memory overhead compared to the control flow checking techniques.

The rest of this paper is organized as follows: in the next section the fault model for this paper is presented, then in section III the proposed technique introduced and analyzed in detail. Section IV shows the fault injection technique, and the experimental results of the proposed method are evaluated in this section. Finally some conclusions are given in section V.

II. FAULT MODEL

Memory faults have various approaches to manifest themselves. There are two parameters which are important to make the decision that if an error will stop the application from its normal behavior or will not affect the application with any discernible harm. The first one is the memory cell which has been influenced by the fault, and the second one is the time at which the fault leave its mark to a program execution. For an instance, the system will not crash if a fault happen in an unused location of its memory. However, if a fault happen in the data or code segment of the program it may cause any kind of errors such as: data error, control flow error, segmentation fault, bus error or illegal instruction. system hardware can detect only segmentation fault, bus error, and illegal instruction and then the operating system will catch them and map them to a signal [14].

Control flow errors coupled with data errors are more likely to happen among the mentioned types of errors. The operating systems are not able to detect these kinds of errors; therefore, it is necessary to detect and correct them by some mechanisms in safety-critical systems.

As mentioned before, in control flow checking techniques, the common approach is to divide the source code into some basic blocks. Having a precise look on these techniques, occurrence of any fault in the system may cause one of the following six types of CFEs [11]:

1- Illegal branches from the end of any basic block to the beginning of another one.
2- Legal but incorrect branches from the end of any basic block to the beginning of another basic block. As mentioned before, this type of CFE is named wrong-successor CFE.
3- A jump from the end of a basic block to any point of another basic block.
4- A jump from any point of a basic block to any point of another basic block.
5- A jump from any point intra a basic block into any point of itself.
6- A jump from any point of a basic block to an unused memory space.

Generally, second type of CFEs which are known as wrong-successor CFEs are due to fault occurrence in data (control variables). For example, Fig. 2 shows a hypothetical scenario. Assume that in this scenario before execution of the "*while*" instruction the value of *x* variable was equal to 5 and a transient fault had happened in the location of this variable in the memory. This fault flipped the fourth bit of *x* from 0 to 1 caused the value of *x* change from 5 to 13 which lead the control flow of the program to execute *block 2* instead of *block 1*. This scenario illustrates the process of occurrence of a wrong-successor CFE.

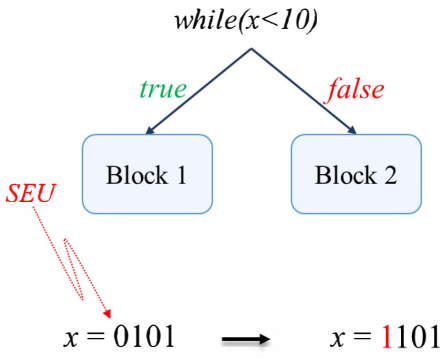

Figure 2. Example of wrong-successor CFE

Due to the proportion of execution time of loop commands compared to the execution time of the program, control variables have a much higher probability of being hit by a fault than variables in instructions that execute only once.

Normally variables which are used in a program are classified into two groups, control variables and non-control variables. A variable that may alter the control flow of the program is a control variable. There are two types of control variables. First, the variables which are used in condition statement of a branch instruction. Second, the variables which are not in any branch instruction directly, still at least the value of one control variable depends on them. In both two cases, the variable affects the control flow of the program directly or indirectly. Fig. 3 shows the matrix multiplication procedure. In this procedure *{i, m, j, p, k, n}* are the control variables and *{a, b, c arrays and sum}* are the non-control variables.

Since the main goal of this paper is to detect wrong-successor CFEs, some faults should be injected into the system to create wrong-successor errors. As mentioned before wrong-successor errors are caused by fault occurrence in control variables. Therefore, in order to evaluate the proposed technique the effect of Single Event Upset (SEU) on control variables must be modeled using fault injection.

III. Proposed Technique

One of the common techniques to detect and correct soft errors is to provide programs with data and instructions replication. Replicating is a method that usually has a huge performance and memory overhead also programming languages have some limitations on supporting these methods. Therefore, replication of all of the data and instructions completely, is not a feasible technique, although these techniques are able to detect high rate of faults. In most of the applications, especially real-time systems, there are some constraints on the memory storage size and the execution time. Hence if these constraints became denied, the performance of the system will be reduced and the proper output will not be achieved.

```
for (i = 0; i < m; i++){
    for (j = 0; j < p; j++){
        sum = 0;
        for (k = 0; k < n; k++)
            sum = sum + a[i][k] * b[k][j];
        c[i][j] = sum;
    }
}
```

Figure 3. Matrix multiplication procedure

The major ideas of this paper in order to detect wrong-successor CFEs, are differentiation of control variables and applying a proper fault masking technique. In this paper, a simple triple modular redundant (TMR) system was chosen as the fault masking technique. Therefore, two clones are being made out of any control variable. The value of the clones will change with the original one at any point of the program that an assignment instruction alters the value of control variable. In the condition clause of every control flow statement of the program such as "*while*" and "*for*" loops, a software voter is applied instead of control variables. The control variable and its clones are the inputs of the TMR's voter. Hiring a voter for control flow statements ensures the correctness of branches. Also in case of error occurrence in any control variable, the voter will correct the faulty value keeping the system safe from error propagation. Fig. 4 shows a fictional piece of code in its left side that is provided with the proposed technique on the right side.

The proposed technique hires an exact majority voter to mask the effect of an error that may happen in a control variable. An exact majority voter with *n* inputs produces result if and only if there is consensus between at least *(n+1)/2* of its inputs otherwise the voter throws an exception and leads the system towards a fail-safe state [15]. Therefore, the exact majority voter in the TMR system masks a fault in any of clones or the control variable.

```
int sum=0, x=0;
while(x<10) {
    sum = sum + x;
    x++;
}
```

Normal

```
int sum=0, x=0, x1=0, x2=0;
while(voting(x, x1, x2)<10) {
    sum = sum + x;
    x++;
    x1++;
    x2++;
}
```

With redundancy

Figure 4. Redundant codes to mask wrong-successor CFEs

In order to achieve a higher performance and reduction of the execution time of the voting, at first, the voter judge only base on its two primary inputs. If these two values be in agreement it means that no error had happened in any of these two values; therefore, selecting anyone of them is a right decision to make. This paper follows the single event upset (SEU) model which states that only one fault may occur in the memory at a time. Therefore, if the two primary inputs of the voter do not go along with each other, it means that the error had happened already so the third input of the voter must be error-free, and it is a thoughtful choice to make the further decision based on the third input of the voter. In fact, this voter always produces results. Finally, the result of the voting spread to its inputs' sources so the program will no longer face with the detected error.

Since the proposed technique is completely software implemented, it does not depend on any specific hardware; therefore, it can be implemented on a system with any kind of hardware configuration. This technique can be attached to any control flow checking technique completing their ability of error detection by masking wrong-successor control flow errors.

IV. Experimental Results

In this section, experimental environment and the benchmarks which are used to evaluate the proposed technique are introduced. Also the process of fault injection is explained in detail. Finally, the results of the experiments on the proposed technique are analyzed.

To evaluate the proposed technique, the experiments were run on a system with a quad core Intel® Core™ i7-3632QM processor and 4GB of RAM as the simulation environment. This system had a Linux Ubuntu 14.04 as its operating system. The proposed technique was implemented by the C programming language with the GNU Compiler Collection (gcc) then it was applied on *dijkstra, FFT, patricia, sha and stringsearch* benchmarks from various categories of the MiBench package [16].

In this paper, the used fault injection method is software implemented fault injection (SWIFI). In order to create wrong-successor CFEs, the faults must be injected to the control variables.

The current study adhered to the single event upset model which states that only one fault may occur during an execution. To inject the fault, the program execution is interrupted by a timer with a random initial value. In this state, the normal execution of the program will be paused and the system will run the interrupt subroutine. In this subroutine, a control variable will be selected randomly then one bit of this control variable will be flipped with the *xor* operator. Consequently, this process leads the system toward a wrong-successor CFE.

The important parameters in the evaluation of redundancy based techniques are performance overhead, memory size overhead and fault detection/correction coverage. The effectiveness of proposed technique in terms of the mentioned parameters is discussed later.

To evaluate the rate of fault masking, the mentioned benchmarks were run on an error-free condition and their output were saved as the golden result. Then redundant codes are added to the benchmarks and 10,000 faults are injected into each benchmark. The result of each test is compared with the golden result. If the test result and the golden result are equal it means that the proposed technique is able to mask the injected fault successfully. In order to ensure the results of the experiments it is necessary to repeat tests for 10,000 times to achieve confidence-level equal to 95% with confidence-interval equal to 1 this means that the results of this experiment, with the probability of 95%, at most have 1% error.

The difference between execution time before and after adding redundant codes is used to calculate performance overhead. In order to calculate memory size overhead, difference between object file sizes is used. Table 1 shows the results of applying the proposed technique on the mentioned benchmarks. As it can be seen in this table, the proposed technique has the ability to mask all the injected faults completely while it has 20.95% performance overhead with 6.13% memory overhead.

Table 1. Experimental results of the proposed technique

Benchmark	CFE Masking (%)	Overheads (%)	
		Performance	Memory
dijkstra	100	15.87	6.2
FFT	100	21.07	6.05
patricia	100	20.42	6.46
sha	100	17.2	4.62
stringsearch	100	30.18	7.33
Average	100	20.95	6.13

V. Conclusions

One of the most important hazards in memories is occurrence of transient faults. These faults may lead to control flow errors or data errors. Regarding to the high rate of control flow errors occurrence, lots of techniques have been proposed to detect or correct this kind of errors. Among all CFEs, a part of them occur due to faults in data which are named wrong-successor CFEs that no control flow checking technique pays attention to them.

The proposed technique in this paper masks the wrong-successor CFEs. The foundation of this technique is to distinguish the control variables from other variables and applying a traditional fault masking technique on them. To evaluate this technique, 10,000 faults were injected on each of the five various benchmarks of the MiBench package. Experiment results show that the injected faults are being masked completely by the proposed technique that have almost 21% performance overhead with 6% memory overhead. This technique is software-implemented and does not need any extra hardware. Also, this technique is able to be applied on any system with no dependency to any specific hardware. Applying this technique on all of the previous control flow checking methods, because of its perfect wrong-successor CFE correction coverage and low overheads, is reasonable and feasible.

REFERENCES

[1] J. C. Knight, "Safety critical systems: challenges and directions", *Proceedings of the 24th International Conference on Software Engineering (ICSE 2002)*, pp. 547-550, 2002.

[2] X. Li, *Exploiting inherent program redundancy for fault tolerance*, ProQuest, 2009.

[3] Y. Sedaghat, S. G. Miremadi, and M. Fazeli, "A software-based error detection technique using encoded signatures", *21st IEEE International Symposium on Defect and Fault Tolerance in VLSI Systems (DFT'06)*, pp. 389-400, 2006.

[4] I. Koren and C. M. Krishna, *Fault-tolerant systems*, Morgan Kaufmann, 2010.

[5] R. C. Baumann, "Soft errors in advanced semiconductor devices: Part I: The three radiation sources", *IEEE Transactions on Device and Materials Reliability,* vol. 1, pp. 17-22, 2001.

[6] T. C. May and M. H. Woods, "Alpha-particle-induced soft errors in dynamic memories", *IEEE Transactions on Electron Devices,* vol. 26, pp. 2-9, 1979.

[7] N. Oh, P. P. Shirvani, and E. J. McCluskey, "Error detection by duplicated instructions in super-scalar processors", *IEEE Transactions on Reliability,* vol. 51, pp. 63-75, 2002.

[8] N. Khoshavi, H. R. Zarandi, and M. Maghsoudloo, "Control-flow error recovery using commodity multi-core architecture features", *17th IEEE International On-Line Testing Symposium (IOLTS)*, pp. 190-191, 2011.

[9] N. F. Ghalaty, M. Fazeli, H. I. Rad, and S. G. Miremadi, "Software-based control flow error detection and correction using branch triplication", *17th IEEE International On-Line Testing Symposium (IOLTS)*, pp. 214-217, 2011.

[10] A. Shrivastava, A. Rhisheekesan, R. Jeyapaul, and C.-J. Wu, "Quantitative analysis of Control Flow Checking mechanisms for soft errors", *51st ACM/EDAC/IEEE Design Automation Conference (DAC)*, pp. 1-6, 2014.

[11] S. Asghari, H. Taheri, and H. Pedram, "Software-based Control Flow Checking against Transient Faults in Industrial Environments", *IEEE Transactions on Industrial Informatics,* vol. 10, pp. 481-490, 2014.

[12] M. Maghsoudloo, H. R. Zarandi, and N. Khoshavi, "An efficient adaptive software-implemented technique to detect control-flow errors in multi-core architectures", *Microelectronics Reliability,* vol. 52, pp. 2812-2828, 2012.

[13] N. Oh, P. P. Shirvani, and E. J. McCluskey, "Control-flow checking by software signatures", *IEEE Transactions on Reliability,* vol. 51, pp. 111-122, 2002.

[14] A. Heinig, M. Engel, F. Schmoll, and P. Marwedel, "Improving transient memory fault resilience of an H. 264 decoder", *ESTImedia*, pp. 121-130, 2010.

[15] M. Rezaee, Y. Sedaghat, and M. Khosravi-Farmad, "A Confidence-based Software Voter for Safety-Critical Systems", *12th IEEE International Conference on Dependable, Autonomic and Secure Computing (DASC)*, pp. 196-201, 2014.

[16] M. R. Guthaus, J. S. Ringenberg, D. Ernst, T. M. Austin, T. Mudge, and R. B. Brown, "MiBench: A free, commercially representative embedded benchmark suite", *IEEE International Workshop on Workload Characterization (WWC-4)*, pp. 3-14, 2001.