Soft-error Detection Using Control Flow Assertions

O. Goloubeva, M. Rebaudengo, M. Sonza Reorda, M. Violante
Politecnico di Torino, Dipartimento di Automatica e Informatica
Torino, Italy

Abstract

Over the last years, an increasing number of safety-critical tasks have been demanded to computer systems. In this paper, a software-based approach for developing safety-critical applications is analyzed. The technique is based on the introduction of additional executable assertions to check the correct execution of the program control flow. By applying the proposed technique, several benchmark applications have been hardened against transient errors. Fault Injection campaigns have been performed to evaluate the fault detection capability of the proposed technique in comparison with state-of-the-art alternative assertion-based methods. Experimental results show that the proposed approach is far more effective than the other considered techniques in terms of fault detection capability, at the cost of a limited increase in memory requirements and in performance overhead.

1. Introduction

The increasing popularity of low-cost safety-critical computer-based applications in several areas, such as automotive or biomedical equipments, asks for the availability of low-cost dependable systems. Transient and intermittent faults due to environmental effects such as electromagnetic interferences, power glitches, or highly energized particles can cause unpredictable behaviors of computer-based systems. For example, Single Event Upsets (SEUs) are transient faults caused by the interaction of highly energized particles with the silicon substrate, which cause the state of a memory cell or a flip-flop to change its logic value; the effects of SEUs on software applications have been deeply analyzed in the literature [1]. This kind of effects is not limited to the space environment, but it has also been observed at the ground level, where very low energy is required to change the state of a storage element in deep sub-micron technology devices.

Fault avoidance techniques, such as radiation hardening or hardware redundancy have been traditionally considered to achieve reliability requirements. Radiation-hardened devices are characterized by a fast obsolescence due to the time required to modify the corresponding commercial device. Hardware redundancy is a viable solution in many different applications, but it is not feasible for those where cost is a critical issue. The need for state-of-the-art high computing performances, coupled with cost containment, provides a strong motivation for investigating feasible alternatives to traditional solutions. The use of Commercial Off-The-Shelf (COTS) components for safety-critical applications has been suggested to accelerate the development cycle and produce cost effective systems. COTS components require specific approaches to take into account the effect of possible hardware faults.

In this paper we consider the effects of faults affecting control flow instructions in microprocessor-based systems. These faults occur when a processor fetches and executes an incorrect instruction during the program execution that modifies the expected control flow.

Various control-flow checking techniques have been proposed in the past to detect such kind of faults. Many of them are based on a watchdog processor [2-5] to compute run-time signatures from the instructions and compare them with the precomputed signatures. These techniques need either additional hardware or modification of the existing hardware and cannot guarantee portability to various platforms. When the hardware cannot be changed, a pure software method is the only feasible solution.

The introduction of *Software Implemented Hardware Fault Tolerance* (SIHFT) [6] techniques for fault detection is applicable to COTS-based devices, providing low-cost solutions for enhancing the reliability of these systems without modifying the hardware. The basic idea of *Control Flow checking* [7-9] is to partition the application program in *basic blocks*, i.e., branch-free parts of code. For each block a deterministic signature is computed and errors can be detected by comparing the run-time signature with a pre-computed one. The Control-Flow checking approach does not require any additional hardware. The main advantage of the method lies in the fact that it can be automatically applied to a high-level source code, thus freeing the programmer from the burden of guaranteeing its correct implementation. The method is completely independent on the underlying hardware, and does not rely on any specific error detection mechanism: the safety of the resulting system only comes from the checks introduced in the code. The main drawbacks are time overhead and code size increase, resulting from the added instructions.

In this paper we propose an enhanced version of the Control Flow Checking technique, called *YACCA* (*Yet Another Control-Flow Checking using Assertions*). The proposed approach exploits the information available in the Program Graph, and during the program execution, for each basic block, it checks if the block is reached from a legal block (according to the Program Graph), otherwise a control flow error is detected. The novelty of the proposed method consists in the adopted technique for the signature generation and control check.

With respect to alternative solutions ([8-9]) our method is able to cover all the faults affecting the branches, except for a very limited class of faults as described in the paper. The proposed rules are particularly focused on reducing the memory size and time overheads without loosing in fault detection capabilities. In order to evaluate the effectiveness of the approach several Fault Injection experiments have been performed on a set of benchmark programs, considering a particular fault model that emulates control flow errors. Such a kind of Fault Injection experiments show the effectiveness of the approach with respect to the state-of-the-art techniques, in terms of fault coverage, memory and performance overheads.

The approach can be easily complemented with other SIHFT techniques (e.g., [10-11]) in order to cover faults affecting data and to guarantee a fault tolerant behavior thanks to the introduction of error correction capabilities.

The remainder of the paper is organized as follows. Section 2 reports an overview of previous Control Flow Checking techniques, while section 3 describes the proposed hardening approach. In Section 4 we analytically demonstrate the error detection properties of the proposed method. Section 5 reports the experimental results we gathered and finally section 6 draws some conclusions.

2. Previous work

Control-Flow Checking approaches are based on a partition of the program code into basic blocks [12]. A basic block is a sequence of consecutive instructions in which, in absence of faults, the flow of control always enters at the beginning and leaves at the end. This means that a basic block does not contain any instructions that may change the control flow, such as jump, branch or call instructions, except for the last one, possibly. Furthermore, no instructions in the basic block can be the destination of a branch, jump or call instruction, except for the first one, possibly.

A program P can be represented with a graph composed of a set of nodes V and a set of edges E, $P=\{V,E\}$ [13], where $V=\{v_1,v_2,\dots,v_i,\dots,v_n\}$ and $E=\{e_1,e_2,\dots,e_i,\dots,e_m\}$.

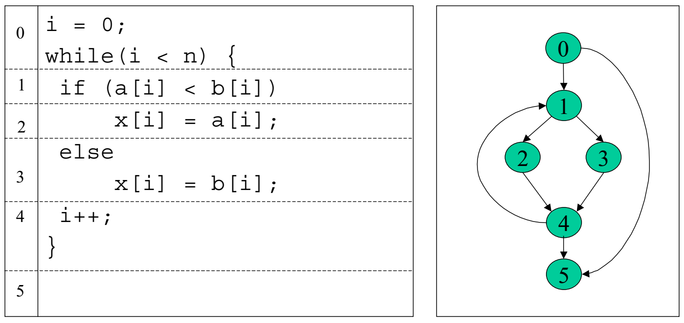
Each node v_i represents a basic block and each edge e_i represents the branch $br_{i,j}$ from v_i to v_j. The edges $br_{i,j}$ are not necessarily explicit branch instructions, but they also represent jumps, subroutine calls, returns instructions. High-level programming languages usually define a specific structure to delimit the borders of each block (e.g., the symbol { in C programming language).

Considering the Program Graph $P=\{V,E\}$, for each node v_i it is possible to define $suc(v_i)$ as the set of nodes successor of v_i and $pred(v_i)$ as the set of nodes predecessor of v_i.

A node v_j belongs to $suc(v_i)$ if and only if $br_{i,j}$ is included in E. Similarly, v_j belongs to $pred(v_i)$ if and only if $br_{j,i}$ is included in E.

Considering a program represented by its Program Graph $P=\{V,E\}$, then during the execution of P, $br_{i,j}$ is *illegal* if $br_{i,j}$ is not included in E. This illegal branch indicates a *Control Flow Error*, which can be caused by transient or permanent faults.

As an example we consider the fragment of a sample routine shown in Fig. 1, where the basic blocks are also numbered in Fig.1.a, and the corresponding Program Graph is shown in Fig. 1.b.



```
0  i = 0;
   while(i < n) {
1   if (a[i] < b[i])
2       x[i] = a[i];
3   else
        x[i] = b[i];
4   i++;
   }
5
```

a)   b)

Fig. 1: Source code and program graph example.

The most important solutions proposed in the literature are the techniques called ECCA (Enhanced Control Flow Checking using Assertions) [8] and CFCSS (Control Flow Checking by Software Signatures) [9].

ECCA assigns a unique prime number identifier to each basic block of a program. A global integer variable is added to check the control flow correct execution. This variable is dynamically updated during the execution. Two lines of code are asserted into each block:

- a *test* assertion is executed at the beginning of the block and checks if the previous basic block is permissible, according to the Program Graph. A divide by zero error identifies the control flow error.
- A *set* assignment is executed at the end of the block and updates the identifier taking into account the whole set of possible next basic blocks.

ECCA is able to detect all the single control flow faults crossing block boundaries, except from the faults that cause an incorrect decision on a conditional branch. ECCA is not able to detect control faults not crossing the block boundaries, i.e., faults that cause a branch into the same block.

CFCSS assigns a unique signature s_i to each basic block. A global variable (called G) contains the run-time signature. In absence of errors, G contains the signature associated to the current basic block. G is initialized with the signature of the first block of the program.

At the beginning of the basic block an additional instruction computes the signature of the destination block from the signature of the source block: G is updated with the EXOR function between the signature of the current node and the destination node. If the control can enter from multiple blocks, then an adjusting signature is assigned in each source block and used in the destination block to compute the signature.

As a limitation, CFCSS cannot cover control flow errors if multiple nodes share multiple nodes as their destination nodes. As described in [9], given a graph with the set of edges $E=\{br_{1,4}, br_{1,5}, br_{2,5}, br_{2,6}, br_{3,5}, br_{3,6}\}$, an illegal branch between node v_1 to node v_6 corresponding to the branch $br_{1,6}$ is not detected by the method.

3. Transformation Rules

YACCA checks the control flow of programs by using a dedicated global integer variable (called *code*), which contains the run-time signature associated with the current node in the program flow graph.

Every basic block is identified by a unique signature defined at the compile time. The following assertions are introduced into each basic block v_i:

- a *test* assertion controls the signature of the previous basic block and checks if it is permissible, according to the Program Graph, i.e., if the entering node v_j belongs to $pred(v_i)$.
- A *set* assignment updates the signature, setting it to the correspondent value B_i.

The main novelties, described in the following subsections, consist in the definition of the test and set assertions. Moreover, differently from other approaches, in each basic block the test and set statements are introduced at the beginning and at the end of the block. This approach allows YACCA to cover all the single control flow faults, including the ones not crossing the block boundaries.

Moreover, the approach also integrates an additional rule defined by the same authors in [10], in order to detect possible faults affecting the decision operand within the conditional branches: for every test statement the test is repeated at the beginning of the target basic block of both the true and (possible) false clause. If the two versions of the test (the original and the newly introduced) produce different results, an error is detected.

A deeper description is reported in the following subsections.

3.1. Test assertion

When the program enters into a basic block v_i arriving from a basic block v_j an assertion checks if the transition is legal, i.e., if the edge $br_{j,i}$ belongs to set E of edges in the Program Graph. Given a node v_i, it is possible to find a set $E_i \subseteq E$, which is composed of the edges corresponding to the branches entering into the node v_i. A constant value, called $PREVIOUS_i$, is computed as the product of the signatures corresponding to the list of nodes entering into the node v_i, according to the following rule:

$PREVIOUS_i = \Pi B_j, \forall v_j$ with a branch $br_{j,i} \in E_i$.

In order to check if the branch is legal, the test assertion verifies if $PREVIOUS_i$ is a multiple of the current signature by means of the following instruction:

```
if (PREVIOUS_i % code) error();
```

From the implementation point of view, since this instruction is particularly critical from the computational point of view (due to the division operation), an equivalent instruction has been introduced based on a list of comparisons:

```
if ((code != B_a) && (code != B_b) && (...) && (code != B_n))
    error();
```

with the set $E_i = \{br_{a,i}, br_{b,i}, \dots, br_{n,i}\}$.

In order to avoid the introduction of additional branches into the code, which may be affected by faults themselves, a further optimization has been introduced. A global flag variable, called *ERR_CODE*, is defined and initialized to 0. The test assertion updates the ERR_CODE variable by the following boolean operation:

ERR_CODE |= ((code != B_a) && (...) && (code != B_n))

with the set $E_i = \{br_{a,i}, br_{b,i}, \dots, br_{n,i}\}$.

3.2. Set assignment

For each basic block v_i YACCA sets the signature to the value corresponding to the current basic block. A trivial solution can be adopted introducing the following instruction:

code = B_i;

The possible faults causing an erroneous branch to this instruction are undetectable. For this reason, the adopted solution sets the signature for the current node v_i by using a function depending on the current signature.

Given a node v_i, the general formula is the following:

code = (code & M1) $\oplus$ M2 (1)

where M1 represents a constant mask depending on the signatures of the nodes belonging to pred(v_i); while M2 represents a constant mask depending both on the signature of the current node and those of the nodes belonging to pred(v_i).

The appropriate values of M1 and M2 must be defined, in order to obtain that the set assignment statement modifies the value of the signature into the proper one, e.g., the signature B_i of the current node v_i.

As an example, given a node v_i, if the set of their predecessors pred(v_i) is composed of a single node v_j, M1 corresponds to the value -1 and M2 corresponds to the operation $B_j \oplus B_i$. In this case the general statement (1) can be written with the following instruction:

code = code $\oplus$ ($B_j \oplus B_i$) (2)

In absence of errors, before the execution of this instruction the value of code corresponds to B_j and after it, the new value corresponds to B_i. We remark that the operation $B_j \oplus B_i$ can be executed at the compile-time since the operands are constant values.

As a second example, if the set of predecessors pred(v_i) is composed of 2 nodes, e.g., v_j and v_k, the statement (1) becomes the following:

code = (code & ($B_j \overline{\oplus} B_k$)) $\oplus$ (B_j & ($B_j \overline{\oplus} B_k$) $\oplus$ B_i) (3)

In absence of errors, before the execution of this instruction the value of code corresponds to B_j or B_k, but in any case, after it, the new value corresponds to B_i. We remark that the operations $B_j \overline{\oplus} B_k$ and $B_j \& (B_j \overline{\oplus} B_k) \oplus B_i$ can be executed at the compile-time since all the operands are constant values.

In order for our method to be effective the mask M1 should never assume the value 0, otherwise an aliasing effect could arise.

The main difference between the *set assignment* proposed here and the one proposed in CFCSS is that in YACCA the signature is computed without any adjustment dependent on the possible multiple fanouts. This problem in CFCSS causes the aliasing effect, which is not present in YACCA.

4. Error detection capabilities

For an analytical evaluation of the detection capabilities of YACCA, given a node v_i, we

define a fault model that considers control flow errors of the following types:
1. a branch to a basic block `v`$_j$ not belonging to `suc(v`$_i$`)`
2. a branch to the begin of a basic block `v`$_j$ belonging to `suc(v`$_i$`)`
3. a branch to somewhere inside a basic block `v`$_j$ belonging to `suc(v`$_i$`)`.

Theorem With the hypothesis that the test and set assertions are executed atomically with a single instruction, YACCA method is capable of detecting all single control flow faults.

Proof

We proof the theorem by showing that all the 3 fault types are detected by YACCA.

Type 1: a branch to a basic block v_j not belonging to suc(v_i)

The test assertion at the begin or at the end of the basic block `v`$_j$ detects the fault.

Type 2: a branch to the begin of a basic block v_j belonging to suc(v_i)

The control flow check introduced at the beginning of the basic block `v`$_j$, according to the SIHFT rules introduced in [10], detects the fault.

Type 3: a branch to somewhere inside a basic block v_j belonging to suc(v_i)

The test and set assertions at the beginning of the block are not executed and the test assertion at the end of the basic block `v`$_j$ detects the fault.

In the real case, the test and set assertions are not executed atomically. As a consequence, the very particular case of a branch to the set assignment at the beginning of a basic block v_j belonging to `suc(v`$_i$`)` is not detected by YACCA, only.

5. Experimental results

In order to assess the effectiveness of the proposed approach, we have performed several fault injection sessions on a system composed of a Sparc V8 microprocessor running 4 benchmark programs implementing the following tasks:

- a 5x5 matrix multiplication (*M*),
- the Kalman Filter (*K*)
- the fifth order elliptical wave filter (*E*)
- the *Lempel Ziv Welch* (LZW) Data Compression algorithm (*L*).

Experiments were performed using an in-house developed emulation-based fault injection environment [14].

We considered 4 versions for each benchmark:

- an un-hardened version
- a safe one, obtained by applying the CFCSS [9] technique to the original code
- a safe one, obtained by applying the ECCA [8] technique to the original code
- a safe one, obtained by applying the YACCA technique to the original code.

By comparing the size and the execution speed of the hardened programs and with respect to the original ones, we recorded the overheads reported in Table 1. These results show that YACCA presents a comparable increase in terms of memory and performance overheads with respect to the CFCSS technique, but it is always better than the ECCA technique.

Moreover, the results reported in Table 1 show that a large difference in terms of overheads is obtained considering the different programs. This is due to the different characteristics of the basic blocks:

- *E* presents basic blocks with many mathematical instructions which are CPU intensive, and the additional instructions are less relevant in terms of size and speed
- *L*, on the other hand, presents many basic blocks with a limited number of instructions.

Program	Memory overhead [%]			Performance overhead [%]		
	CFCSS	ECCA	YACCA	CFCSS	ECCA	YACCA
M	261	408	191	135	199	147
E	124	153	129	107	120	110
K	164	282	217	117	168	156
L	338	630	496	185	426	354

Table 1: Overhead comparison.

The results we gathered during fault injection experiments are reported in Table 2 and 3 where transients faults injected in the un-hardened programs are categorized according to their effects (Table 2) and then compared with those injected in the 3 safe versions (CFCSS, ECCA and YACCA), as reported in Table 3. Fault effects are classified as follows:

- *Effect-less* (*EL*): the fault does not modify the results produced by the program
- *Software Detection* (*SD*): the fault is detected by the software assertion introduced by the code-hardening technique
- *EDM Detection* (*EDM*): the fault is detected by an Error Detection Mechanism embedded into the processor (e.g., illegal instruction, illegal address, software trap, floating point exception, etc.).
- *Wrong answer* (*WA*): the fault modifies the results of the program without being detected
- *Time-out* (*TO*): due to the fault, the program does not end within a given amount of time.

Program	Injected Faults	EL		EDM		WA		TO	
		[#]	[%]	[#]	[%]	[#]	[%]	[#]	[%]
M	5000	274	5.5	2494	49.9	1032	20.6	1200	24.0
E	5000	389	7.8	2814	56.3	540	10.8	1257	25.1
K	5000	615	12.3	2794	55.9	573	11.5	1018	20.4
L	1000	223	22.3	518	51.8	259	25.9	0	0.0

Table 2: Fault Injection experiments for the original version of the benchmark programs.

The adopted fault model is the one described in Section 4: we injected randomly selected bit-flips in the immediate operands of the branch instructions.

Considering the whole set of 16 case studies, the time needed to execute the complete Fault Injection campaign has been 20 hours.

Prog	Faults [#]	CFCSS					ECCA					YACCA				
		EL	SD	EDM	WA	TO	EL	SD	EDM	WA	TO	EL	SD	EDM	WA	TO
M	5000	3.8	53.5	12.8	19.1	10.5	28.4	49.9	7.3	3.7	10.6	4.1	56.0	14.2	0.9	24.5
E	5000	8.8	22.0	33.4	18.7	16.9	24.4	39.8	14.0	4.1	17.5	1.8	54.5	7.6	0.0	35.9
K	5000	10.2	42.4	35.2	1.5	10.6	27.3	42.8	21.3	2.2	6.1	32.6	22.2	31.5	0.4	13.2
L	1000	17.0	44.5	28.5	6.0	4.0	37.5	42.6	17.6	0.6	1.7	42.0	21.1	32.1	0.1	4.7

Table 3: Fault Injection experiments for the hardened versions of the benchmark programs (figures are in percentage unless the number of injected faults)

The results reported in Table 3 demonstrate the effectiveness of the YACCA method as far as fault coverage is considered. A very limited number of control flow faults cause a failure, and the method shows itself to be more powerful than the state-of-the-art alternative approaches.

Note that the experimental results obtained considering the CFCSS method present a higher percentage of wrong answers than the one published in [9]. This is mainly due to the following motivations:

1. we applied the CFCSS technique on the high-level source code, differently from the published results, which are obtained applying the rules on the assembly-level code.

2. 2 different fault models are adopted: they present different characteristics motivating different figures.

6. Conclusions

We presented a new SIHFT technique for the on-line detection of control flow faults. Our method is particularly suited for safety-critical applications implemented by low-cost embedded systems; in these systems memory availability and execution speed are often not a major concern. Starting from a statical analysis based on the program graph construction, the proposed method introduces additional instructions. The method exploits the information available in a Program Graph representation. The main novelties of the proposed method consist in the adopted technique for the signature generation and control check.

A Fault Injection campaign on a representative set of benchmark programs demonstrated the effectiveness of the approach in terms of fault detection, which is always higher than the one achieved considering the alternative state-of-the-art approaches. On the other hand, the method presents memory and performance overheads comparable with those of previous techniques.

7. References

[1] A. Benso, S. Di Carlo, G. Di Natale, P. Prinetto, *Static Analysis of SEU Effects on Software Applications*, IEEE International Test Conference, 2002, pp. 500-508

[2] N.R. Saxena, E.J. McCluskey, *Control Flow Checking Using Watchdog Assists and Extended-Precision Checksums*, IEEE Transactions on Computers, Vol. 39, No. 4, Apr. 1990, pp. 554-559

[3] G. Miremadi, J. Ohlsson, M. Rimen, J. Karlsson, *Use of Time and Address Signatures for Control Flow Checking*, International Conference on Dependable Computing for Critical Applications (DCCA-5), 1995, pp. 113-124

[4] J. Ohlsson, M. Rimen, *Implicit Signature Checking*, 25th International Symposium on Fault-Tolerant Computing, 1995, pp. 218-227

[5] B. Ramamurthy, S. Upadhyaya, *Watchdog processor-assisted fast recovery in distributed systems*, International Conference on Dependable Computing for Critical Applications (DCCA-5), 1995, pp. 125-134

[6] D.K. Pradhan, *Fault-Tolerant Computer System Design*, Prentice Hall PTR, 1996

[7] G. Miremadi, J. Karlsson, U. Gunneflo, J. Torin, *Two Software Techniques for On-line Error Detection*, International Symposium Fault-Tolerant Computing, 1992, pp. 328-335

[8] Z. Alkhalifa, V.S.S. Nair, N. Krishnamurthy, J.A. Abraham, *Design and Evaluation of System-Level Checks for On-Line Control Flow Error Detection*, IEEE Transactions Parallel and Distributed Systems, Vol. 10, No. 6, June 1999, pp. 627-641

[9] N. Oh, P.P. Shirvani, E.J. McCluskey, *Control-Flow Checking by Software Signatures*, IEEE Transactions on Reliability, Vol. 51, No. 2, March 2002, pp. 111-122

[10] P. Cheynet, B. Nicolescu, R. Velazco, M. Rebaudengo, M. Sonza Reorda, M. Violante, *Experimentally evaluating an automatic approach for generating safety-critical software with respect to transient errors*, IEEE Transactions on Nuclear Science, Vol. 47, No. 6, December 2000, pp. 2231-2236

[11] M. Rebaudengo, M. Sonza Reorda, M. Violante, *A New Software-based technique for low-cost Fault-Tolerant application*, IEEE Annual Reliability and Maintainability Symposium, 2003, pp. 25-28

[12] A. Aho, R. Sethi, J. Ullman, *Compilers: Principles, Techniques and Tools*, Addison-Wesley, 1986

[13] S.S. Yau, F.-C. Chen, *An Approach to Concurrent Control Flow Checking*, IEEE Transactions on Software Engineering, Vol. 6, No. 2, March 1980, pp. 126-137

[14] P. L. Civera, L. Macchiarulo, M. Rebaudengo, M. Sonza Reorda, M. Violante, *Exploiting circuit emulation for fast hardness evaluation*, IEEE Transactions on Nuclear Science, Vol. 48, No. 6, Dec. 2001, pp. 2210 -2216