Static analysis of SEU effects on software applications

Static analysis of SEU effects on software applications

Authors: Benso A., Di Carlo S., Di Natale G., Prinetto P.,

Published in the Proceedings of the IEEE International Test Conference (ITC), 7-10 Oct. 2002, Baltimore (MD), USA.

DOI: 10.1109/TEST.2002.1041800

STATIC ANALYSIS OF SEU EFFECTS ON SOFTWARE APPLICATIONS

A. Benso, S. Di Carlo, G. Di Natale, P. Prinetto

Politecnico di Torino
Dipartimento di Automatica e Informatica
Corso Duca degli Abruzzi 24, I-10129, Torino, Italy
Email: {benso, dicarlo, dinatale, prinetto}@polito.it
Webpage: www.testgroup.polito.it

Abstract

Control Flow Errors have been widely addressed in literature as a possible threat to the dependability of computer systems, and many clever techniques have been proposed to detect and tolerate them. Nevertheless, it has never been discussed if the overheads introduced by many of these techniques are justified by a reasonable probability of incurring Control Flow Errors. This paper presents a static executable code analysis methodology able to compute, depending on the target microprocessor platform, the upper-bound probability that a given application incurs in a Control Flow Error.

1 Introduction

It is accepted that large software systems cannot be fault-free. Some faults may be attributed to inaccuracy during the development, while others can come from external causes such as environmental stress.

Transient faults can cause abnormal behaviors of computer systems. Radiations, electromagnetic interference and power glitches are some of the causes of transient faults. For example, in radiation environment, alpha-particles, cosmic rays and solar wind flux can cause a *single event upset* (SEU), which causes the state of a memory cell or a sequential element (e.g., flip-flop) to change from 0 to 1 or from 1 to 0. Recently, the problem is also concerning consumer products that, thanks to device geometries of 0.18 µm and below operating at Vdd of 1.5 volts where less energy is required to change the state of each flip-flop and memory cell, are becoming highly susceptible to transient faults at the ground level.

Fault may affect both data and code of the application. In any case, after a fault has accidentally occurred, the result of a program is unpredictable: for example, a desired function may not be executed, functions might not be performed in the right order, or the program crashes. However, in many situations, the most deceptive event is when the program reaches the end producing a wrong answer which, unfortunately, looks reasonable; in this case, there is no way for the user to establish the reliability of the output. Unlike performance, the reliability of a computer system cannot be evaluated through the use of benchmark programs and standard test methodologies, only, but requires observing the system behavior when a fault appears into the system. Since the MTBF (*Mean Time Between Failure*) in a computer system can be of the order of years, fault occurrence has to be artificially accelerated in order to observe the system behavior under faults without waiting for the natural appearance of actual faults. In many cases, Fault Injection [1] emerged as a viable and effective solution, and has been deeply investigated by both academia and industry [2], [3], [4]. For each Fault Injection experiment it is necessary to select a fault location, an injection time, a fault duration, and the input stimuli for the application. A Fault Injection campaign is able to characterize the application reliability only for a subset of its possible execution flows and therefore the "space" of possible faults and input stimuli makes the execution of a statistically significant number of experiments very difficult.

In this paper we present a new methodology, named Fault Effect Analysis (FEA), able to characterize the probabilities of the possible behaviors of a given application affected by a transient fault in the code. The application code is not actually executed as in Fault Injection experiments, but statically analyzed. This makes possible to analyze all the possible execution paths at once, and allows obtaining more general results about the application reliability with a computational effort drastically lower than Fault Injection techniques.

The proposed technique has been designed to analyze only the effects of faults appearing in the code of the application. The extension to data faults is under way, but out of the scope of this paper.

Experimental results have been obtained implementing a prototypical tool named FEAR (Fault Effect Analysis instRument), able to analyze the executable code of an application compiled for any platforms whose instruction set has been described using an ad-hoc Instruction Set Description Language (ISDL).

The paper is organized as follows: Section 2 points out the motivations behind our technique; Section 3 introduces the target fault model and Section 4 details the proposed Fault Effect Analysis. Section 5 discusses the main differences with Fault Injection techniques. Our prototype tool and some interesting experimental results are presented in Sections 6 and 7. Conclusions and future activities are summarized in Section 8.

2 Motivations

A correct control flow is a fundamental part of correct execution of computer programs. Generally, the techniques used to check the correct sequencing of the instructions (control-flow checking techniques) are based on Signature Analysis, in which a signature associated with a block of instructions is calculated and saved during compile time; then, the same signature is generated during run time and compared with the saved one. It has to be pointed out that most control-flow checking techniques are designed to detect illegal execution flows (i.e., execution flows not present in the application control-flow graph), but fail in detecting erroneous execution flows (e.g., a legal execution flow in which the wrong branch of an "`if`" instruction is executed).

When the hardware design is fixed and cannot be changed, as in Commercial-Off-The-Shelf (COTS) based systems, control-flow error detection can only be achieved by pure software methods. Software Implemented Hardware Fault Tolerance (SIHFT) exploits pure software techniques to detect and/or tolerate faults appearing in the hardware. The main benefit of SIHFT is that it allows for improving the availability of the system without introducing any hardware overhead. Examples of software methods include assertions [5][6], watchdog task [6], Block Signature Self-Checking (BSSC) [7], Error Capturing Instructions (ECI) [7], timers [8], regular-expressions [9], Available Resource-driven Control-flow monitoring (ARC) [10], temporal redundancy methods [11], and System Level Checks [12].

Both hardware- and software-based approaches usually introduce considerable overheads, especially in the system's performances, that often make control flow checking techniques very difficult to be applied in real applications. Nevertheless, it has never been carefully discussed if the overheads introduced by control flow checking techniques are justified by a reasonable probability of incurring Control Flow Errors (CFE).

We addressed this problem defining a new Fault Effect Analysis (FEA) methodology able to characterize, depending on the target microprocessor platform, the probabilities of the possible behaviors of a given application affected by a transient fault in the code. The proposed FEA is performed exhaustively considering SEU faults in every bit of the application code and of those data that may affect the application flow. The application code is not actually executed as in Fault Injection experiments, but statically analyzed in order to compute the probabilities of the application's possible behaviors. The goal of the presented technique is therefore to allow the software engineer to more deeply understand the possible behaviors and weaknesses of a software application whose code has been affected by a fault in the system hardware.

3 Fault Model

In this research we analyze errors caused by a single bit-flip, or SEU, appearing in the code of the target application. It is important to point out that we are interested only in the effect of the fault (i.e., the corruption of one or more instructions in the execution flow), and not in the physical location of the fault itself. Therefore, faults appearing in the code segment of the memory, in the system or internal buses, in the cache, or in the microprocessor fetch or decoding unit, are in our research functionally equivalent.

The possible scenarios generated by a fault appearing in the code of an application can be formalized as follows:

- *Misinterpretation of an instruction operand*: the hit instruction is correctly interpreted, its length is unchanged, but the value of one of the instruction operands (e.g., Immediate Data, Register, Addressing mode, …) is different. Figure 1a shows an example of the consequences of an error affecting the interpretation of an instruction operand;
- *Misinterpretation of a single instruction*: the fault transforms an instruction into another one of the same length; the following instructions remain untouched (Figure 1b) and are correctly interpreted;
- *Misinterpretation of a set of instructions*: if the fault modifies a bit of a field whose value affects the instruction type and length, the error propagates along the following lines of code, causing a new and different sequence of instructions to be executed.

In the following, we will refer to the sequence of misinterpreted instructions as Realignment Sequence. The Realignment Sequence can terminate either because one of

its instructions is illegal and therefore triggers an exception, or because, after a certain number of instructions, the Instruction Pointer re-aligns to a legal instruction sequence. Figure 1c shows an example of a Realignment Sequence caused by a single bit error.

Considering the mentioned scenarios, the possible fault effects can be grouped in the following *termination events*:

- *Illegal Instruction*: the hit instruction or one instruction in the Realignment Sequence is illegal. The program is terminated by an Illegal Instruction Exception;
- *System Exception*: the hit instruction and all the instructions in the Realignment Sequence are legal, but one of them tries to perform an illegal operation (e.g., a division by zero, or an illegal memory reference). The program is terminated by a System Exception;
- *Normal Termination (Undefined result)*: the hit instruction and all the instructions in the Realignment Sequence are legal, the Instruction Pointer re-aligns to the original sequence, and the program terminates normally.

Whereas the first two cases are less critical since the error is detected by the system, the last case is in many cases the most feared one, since the error is not detected and there is no way of guaranteeing the correctness of the results. If the result is faulty, the situation is usually referred to as *Fail Silent Violation*, otherwise the fault is said to be *Fail Silent*.

The main goal of the fault effect analysis presented in this paper is to analytically study the behavior of the target application when a fault appearing in the code does not lead to the triggering of an exception.

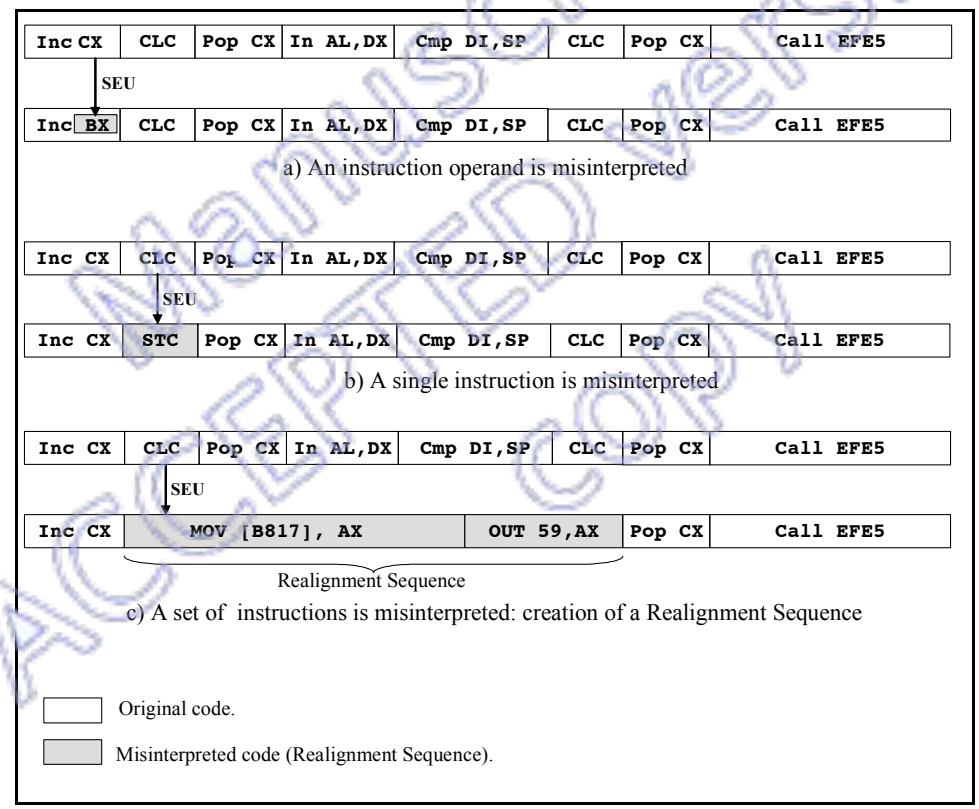

Figure 1: Consequences of a single bit error on the instruction decoding.

4 Fault Effect Analysis

Before detailing the proposed Fault Effect Analysis we introduce some basic definitions needed to clarify the following concepts.

A program can be considered as a sequence of instructions, and the execution of the program can be viewed as executing instructions in a desired sequence. Inside this sequence a *basic block* can be defined as a set of instructions without any branching inside or outside except

for the last one. Using the concept of basic blocks, a program can be therefore associated to a *Control Flow Graph*, which consists of basic blocks and directed edges connecting basic blocks, and which represents all the possible and legal execution flows of the target application. Each execution of the program can be mapped in one visit of the graph. A *Control Flow Error* (CFE) occurs when the execution flow does not match any legal traversing of the control flow graph. We call *Control Block Errors* (CBE), those errors that modify one instruction either into another instruction or into a Realignment Sequence without modifying the program control flow. All Control Flow checking techniques mentioned in the introduction aim at reducing the number of Fail Silent Violations caused by Control Flow Errors (i.e., the execution flows that reach the Normal Termination following an illegal flow of operations and producing incorrect results).

To assess the probability of a Fail Silent Violation caused by a CFE, we have to deeply analyze the consequences an error can have on the program flow. To do so, we statically perform an FEA on the application executable code.

The execution flow of an application is determined by control flow instructions like *jump*, *call*, or rets. From now on we will refer to this type of instructions as *jump instructions*. There are *direct jumps*, where the destination address is fixed and expressed as immediate data, and *indirect jumps*, where the destination address is contained in a register, in the data memory, or in the stack.

The jump instructions determine the program flow and, in general, if an error involves a jump instruction we have a CFE and, potentially, a Fail Silent Violation. There are in fact two possibilities:

- The error causes the misinterpretation of a jump instruction in the original code sequence.
- A new jump instruction is erroneously interpreted in the hit instruction or in the Realignment Sequence.

Figure 2 presents the Control Flow Analysis performed to obtain the results that will be presented in Section 7.

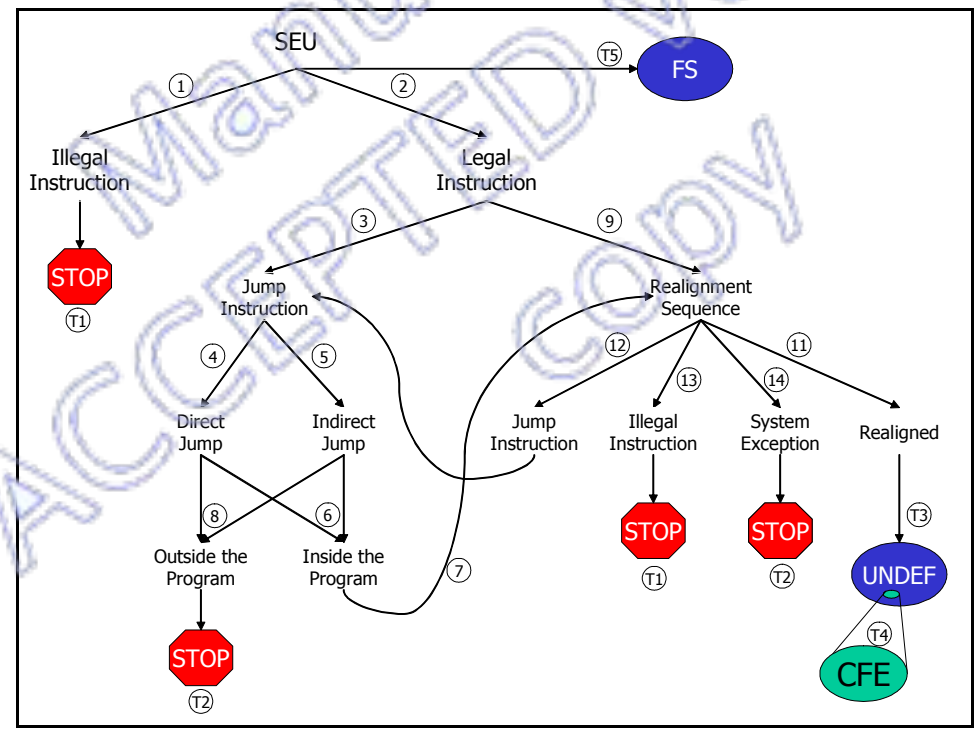

Figure 2: FEA on the program flow.

In the analysis tree showed in Figure 2, the effect of a SEU can evolve into one of the *termination events* described before in Section 3. After its appearance, a SEU can transform the affected instruction in an illegal instruction (1), a legal misinterpretation of the current instruction (2), or the same instruction, due to *don't care* bits in the instruction opcode. In the latter case, the behavior is classified as a Fail Silent (FS) (T5). Whereas

the first case will trigger an illegal exception (T1), in the second case, if the hit instruction is interpreted as a jump (3), it can cause a direct jump (4) or an indirect one (5). In the first case (4), since the destination address is contained in the code as an immediate field, it is possible to follow the program flow. There are two possibilities:

- The destination address points to a byte *internal* to the program code (6): in this case, it is possible to continue the analysis starting from the destination address. Since the destination address not necessarily points to the first byte of an instruction, a Realignment Sequence could be started (7).
- The destination address points to a byte *external* to the program code (8): in this case, the Instruction Pointer will be driven in a portion of the memory outside the program, and it is not possible to continue the analysis. Usually the access to the external memory address is detected and a processor exception occurs (T2).

If the hit instruction is misinterpreted as an indirect jump (5), it is impossible to discover the destination address, since it will depend on data related to the program execution in the moment of the error. In this case, if the size of the register containing the destination address (MD) and the dimension of the program code (CD) are known, it is possible to calculate the probability of jumping inside the code (*Internal Jumping Probability*, IJP). The IJP is given by the following expression:

$$IJP = \frac{CD}{2^{MD}}$$

Given the IJP, it is possible to continue the analysis. Nevertheless, in this case it is not possible to know the destination address and it is therefore not possible to continue the analysis on a particular Realignment Sequence. We analyze the entire application code and decode all the possible Realignment Sequences starting from each byte of the code. With this information we can compute a general probability of reaching the termination events T1 and T4 for an indirect jump instruction inside the code segment.

When the hit instruction is not reinterpreted as a jump instruction, a Realignment Sequence is started (9); the following four situations are possible:

- The Realignment Sequence ends because the Instruction Pointer realigned to the original code (11): in this case the program flow returns on a valid sequence, but a different set of instructions (the Realignment Sequence) has been executed instead of the expected one: the program result is unpredictable (T3). This is the termination event that includes the CFEs that might cause a Fail Silent Violation (T4) and that are the real target of the Control Flow Checking techniques referred to in the introduction. The proposed control flow analysis distinguish the upper-bound probability of incurring in CFEs and CBEs, thus giving the software engineer a mean to evaluate the trade-off between overheads introduced by Control-Flow-Checking techniques and the gained dependability of the system;
- A jump instruction is found before the end of the Realignment Sequence (12). In this case, the behavior depends on the type of jump, and the analysis continues as described before;
- An illegal instruction is detected before either the realignment of the Realignment Sequence or the decoding of a "jump" instruction (13). In this case the system triggers an illegal exception (T1);
- An illegal operation is performed by one legal instruction of the Realignment Sequence (14) (e.g., a division by zero, or an instruction addressing memory areas outside the application scope); a System exception is triggered (T2).

The analysis also takes into account that a jump instruction may be *unconditioned* (the jump always occurs), or *conditioned* (the value of a flag determines if the jump is taken). In the first case the analysis has to simply follow the appropriate arrow of Figure 2. In the second case, where the jump is taken depending on run-time data not directly obtainable in the code, a jumping probability is computed. Since the jump condition is usually determined by binary flags, we considered a 50% probability of taking the jump when a conditional jump is encountered. In this case, the analysis continues following both branches and considering their execution with a halved probability. We know that this might be a limitation of our approach, because during the application execution the probabilities of each branch are likely not to be the same.

5 A-FEA vs Fault Injection

The results calculated by the proposed analysis can be very different from the ones obtained using traditional dependability evaluation techniques. In Fault Injection, each experiment requires to select a fault model (location, injection time, duration), and a set of input data for the application. It is not obvious that the chosen input data will traverse all the possible application execution flows. Therefore a Fault Injection campaign is able to characterize the application reliability only for a subset of its possible execution flows and therefore the "space" of possible faults and input stimuli makes the execution of a statistically significant number of experiments very difficult.

We can explain the difference with the following example. Let's consider an application with the control flow graph presented in Figure 3, and let's assume to define a Fault Injection campaign injecting 1,000 random faults in the code segment of the application with a set of input stimuli that will cause the application to execute the following flow: `Begin-A-End`. Since the block `B` of instructions is never executed, at the end of the campaign we can expect all the faults injected in `B` (~50%) to be Fail Silent.

On the contrary, the proposed A-FEA will analyze all the possible execution flows, providing at the end of the analysis a probabilistic figure of the possible behaviors of the application considering the probability of executing each of the branches equal to 0,5.

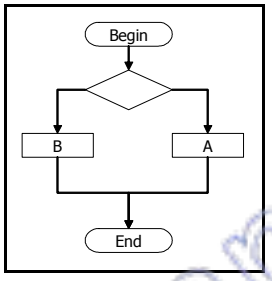

Figure 3: FEA on the program flow

6 Tool

The fault effect analysis methodology presented in the previous section is being automated in FEAR (Fault Effect Analysis instRument). As shown in Figure 4, FEAR reads the application executable code, and an ISDL file containing the detailed description of the instruction set of the target processor. The ISDL file is divided in two parts, one describing the instruction's general format, and the other listing all the instructions implemented in the target microprocessor. The executable file has to be compiled for the same proce or described in the ISDL file. This feature will allow us to evaluate whether or not the FEA is affected by the instruction set format. We currently described in our ISDL the Intel IA-32 (CISC) instruction set, which includes all the instructions from the Intel 286 to the Pentium III processor, and the SPARC V (RISC) instruction set.

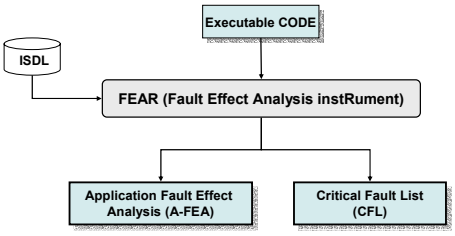

Figure 4: FEAR Input/Output files

FEAR generates two output files:

- The A-FEA is the most important file and stores the SEU effect analysis performed on the application executable code. This file stores the probabilities of each termination event for the considered application.
- The CFL file contains the list of *locations* in the program code or variables where a SEU produces an undefined effect w.r.t. the program flow. This list can be used to generate a reduced fault list to be used in a Fault Injection aiming at more deeply understanding the behavior of the application under the occurrence of SEU faults.

In the current version of FEAR, we are not able to deal with dynamic libraries like dlls. To perform the following experiments we solved the problem by generating executables files including all the required libraries.

7 Experimental Results

The experimental results presented in this Section have been obtained by running FEAR on two SPEC benchmarks (www.spec.org). The first, Dryston, is a mathematical benchmark working on integer numbers. The second, EON, is a 3D studio rendering engine.

We present two sets of experimental results. The first reports the results of the A-FEA on the two benchmarks compiled for the Intel Pentium instruction set. The second highlights the differences between RISC and CISC instruction sets by comparing the A-FEA run on the two benchmarks compiled for both Intel Pentium and Sun Sparc architectures.

In analyzing the presented results, some additional considerations have to be done:

- The analysis assumes that the SEU occurrence probability is equal to 100%. In a real case, this probability depends on the technology, on the working environment (space, ground level, etc), and on the ratio between the code size and the overall system memory.

- The analysis also assumes that each SEU is always activated; nevertheless, in a real execution, many parts of the program are never be executed or are executed for the last time before the SEU appearance. Therefore, a SEU in one of these parts does not affect the correctness of the program.

7.1 A-FEA for Intel Pentium

The first set of experimental results concerns the probabilistic distribution of the termination events, computed applying the A-FEA described in Section 4. Besides the termination events previously introduced, the tool stops the analysis when the SEU effect causes an infinite loop in the application flow. The termination event probabilities for the two benchmarks compiled for the Intel Pentium instruction set are reported in Table 1 (the termination event code introduced in Section 4 is reported in brackets).

	Drystone	*EON*
Illegal (T1)	1,4%	1,6%
Data fault (T2)	35,3%	43,9%
Jump Outside (T2)	2,2%	2,6%
Fail Silent (T3)	26,0%	7%
CBE (T3)	33,49%	43,39%
CFE (T4)	1,7%	1,6%
Loops	0,01%	0,01%

Table 1: Termination Events Probability

Let's comment separately the result of each termination event:

- *Illegal (T1)*: the percentage of illegal instructions is quite low. This result strongly depends on the used instruction set. The Intel Pentium instruction set has many don't care bits in the instruction opcodes, and therefore many SEUs are Fail Silent.
- *Data fault (T2)*: both benchmarks showed a high percentage of data faults, i.e., possible accesses to memory locations outside the data segment. The result is particularly high because the probability of having into a Realignment Sequence an instruction accessing the memory is very high. Being in a Realignment Sequence, the memory address is unknown (or random). We computed the probability of having a legal memory access as $\frac{DS}{2^{MD}}$ where MD is the size of the register containing the memory address and DS the dimension of the data segment. This probability depends on the target processor, the application itself, and the operating system. In our experiment setup the probability is very low and therefore the Data fault probability very high.
- *Jump Outside (T2)*: the probability of having a jump instruction in a Realignment Sequence is much lower than the probability of having a random memory access. Therefore, the probability of jumping outside is quite low.
- *Fail Silent (T3)*: a fault is Fail Silent if it doesn't cause any error. This can happen when injecting a fault into a don't care bit, or into padding bytes, never executed by the application.
- *CBE (T3)*: a fault causes a CBE when the flow within a basic block is affected, but the global execution flow is still legal. This class of errors is very difficult to detect using traditional control-flow error techniques. The results in Table 1 show that this is a very critical class of errors that can lead to unpredictable results.
- *CFE (T4)*: one of the most interesting results is the low percentage of CFE (1,7% and 1,6%). This result seems to point out that CFEs have to be targeted only when very high reliability is required. In all other cases, other classes of errors should be addressed first.

7.2 A-FEA in RISC and CISC

Table 2 and 3 present a comparison obtained running the A-FEA on the two benchmarks compiled for CISC and RISC (Sun Sparc) instruction sets. Results show that the application behavior (and therefore dependability) is strongly related to the instruction set implemented by the target platform. The most interesting difference in the A-FEA analysis is that a fault into a RISC application *never produces a Realignment Sequence*.

	CISC (Intel)	*RISC (Sparc)*
Illegal (T1)	1,4%	8,0%
Data fault (T2)	35,3%	9,4%
Jump Outside (T2)	2,2%	8,75%
Fail Silent (T3)	26,0%	8,74%
CBE (T3)	33,49%	63,0%
CFE (T4)	1,7%	1,65%
Loops	0,01%	0,46%

Table 2: Termination Events Probabilities in CISC and RISC architectures for the Drystone benchmark

	CISC (Intel)	*RISC (Sparc)*
Illegal (T1)	1,6%	9,3%
Data fault (T2)	43,9%	11,5%
Jump Outside (T2)	2,6%	8,2%
Fail Silent (T3)	7%	6,06%
CBE (T3)	43,39%	62,06%
CFE (T4)	1,6%	2,24%
Loops	0,01%	0,64%

Table 3: Termination Events Probabilities in CISC and RISC architectures for the EON benchmark

In the following we analyze each class separately:

- *Illegal (T1)*: the number of faults generating an illegal instruction exception is higher in the RISC architecture because of the different coding scheme of the instruction set.
- *Data fault (T2)*: the number of Data faults is lower in the RISC architecture because the RISC instruction set privileges the use of registers instead of memory locations. Moreover, realignment sequences are never generated in a RISC architecture, and therefore the probability of having a fault generating a memory access instruction is lower than for the CISC architecture.
- *Jump Outside (T2)*: as for the CISC, also for the RISC Instruction set the percentage of Jump Outside seems more related to the instruction set structure than to the application itself.
- *Fail Silent (T3)*: the results suggest that the number of Fail Silent faults is independent from the application and is more a property of the instruction set itself.
- *CBE (T3)*: the fact that in a RISC architecture we do not have Realignment Sequences increases the number of CBEs.
- *CFE ( )*: the results show that the selected instruction set does not influenced the percentage of CFEs. This consideration is quite interesting because it seems to suggest that results obtained using control-flow checking techniques are independent from the target microprocessor.
- *Loops*: in this case the difference between the two architectures is of two orders of magnitude. This suggests that the probability of incurring in a loop is strongly related to the structure of the instruction set.

8 Conclusions and future work

In this paper we presented a Fault Effect Analysis (FEA) methodology aimed at characterizing the behavior of a target application when a SEU occurs in its code segment. The FEA works statically on the application code without actually executing the application, but only probabilistically following all its possible flows. This and other significant differences between the proposed technique and other dependability evaluation techniques like Fault Injection have been discussed. We presented several experimental results obtained running FEA on two complex benchmarks compiled for a CISC and a RISC instruction set. Some results showed a considerable difference between RISC and CISC machines, suggesting that the choice of the target instruction set can be an important parameter that can contribute to the overall system dependability.

Our future activities are focused on the refinement of the technique and in particular in:

- defining a method to evaluate the different execution flows, in order to weight our results with their actual execution probability.
- extending the analysis to other instruction sets in order to more deeply investigate their correlation with the system overall dependability and, in the long run, to propose a new dependability-oriented instruction set.

9 References

[1] J. Clark, D. Pradhan, Fault Injection: A method for Validating Computer-System Dependability, IEEE Computer, June 1995, pp. 47-56

[2] T.A. Delong, B.W. Johnson, J.A. Profeta III, A Fault Injection Technique for VHDL Behavioral-Level Models, IEEE Design & Test of Computers, Winter 1996, pp. 24-33

[3] G.A. Kanawati, N.A. Kanawati, J.A. Abraham, FERRARI: A Flexible Software-Based Fault and Error Injection System, IEEE Trans. on Computers, Vol 44, N. 2, February 1995, pp. 248-260

[4] J. Arlat, M. Aguera, L. Amat, Y. Crouzet, J.C. Fabre, J.-C. Laprie, E. Martins, D. Powell, Fault Injection for Dependability Validation: A Methodology and some Applications, IEEE Transactions on Software Engineering, Vol. 16, No. 2, February 1990

[5] Andrews, D., "Using executable assertions for testing and fault tolerance," 9th Fault-Tolerance Computing Symp., Madison, WI, June 20-22, 1979.

[6] Ersoz, A., D. M. Andrews, and E. J. McCluskey, "The Watchdog Task: Concurrent Error Detection Using Assertions," Stanford University, Center for Reliable Computing, TR 85-8.

[7] Madeira, H. and J. G. Silvia, "On-line Signature Learning and Checking," Dependable Computing for Critical Applications 2, Springer-Verlag, J. F. and R. D. Schlichting (eds), pp. 395-420, 1992.

[8] Madeira, H., M. Rela, and J. G. Silvia, "Time Behavior Monitoring as an Error Detection Mechanism," Dependable Computing for Critical Applications 2, Springer-Verlag, J. F. and R. D. Schlichting (eds), pp. 395-420, Feb. 1993.

[9] A. Benso, S. Di Carlo, G. Di Natale, P. Prinetto, L. Tagliaferri, Control-Flow Checking Via Regular Expressions, IEEE Asian Test Symposium (ATS 2001), Kyoto (J), November 2001

[10] Schuette, M. A., J. P. Shen, "Exploiting Instruction-Level Parallelism for Integrated Control-Flow Monitoring," IEEE Trans. on Computers, Vol. 43, No. 2, pp.129-140, Feb. 1994

[11] Ignatushchenko, V. V., et al., "Effectiveness of temporal redundancy of parallel computational processes," Automation and Remote Control, Vol. 55, No. 6, pt. 2, pp. 900-911, June 1994.

[12] 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 on Parallel and Distributed Systems, Vol. 10, N. 6, June 1999, pp. 627-641