

# DATA CRITICALITY ESTIMATION IN SOFTWARE APPLICATIONS

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

Politecnico di Torino

Dipartimento di Automatica e Informatica

Corso Duca Degli Abruzzi 24, I-10129, Torino, Italy

Email: {benso,dicarlo,dinatale,prinetto,tagliaferri}@polito.it

## Abstract

*In safety-critical applications it is often possible to exploit software techniques to increase system's fault-tolerance. Common approaches are based on data redundancy to prevent data corruption during the software execution. Duplicating most critical variables only can significantly reduce the memory and performance overheads, while still guaranteeing very good results in terms of fault-tolerance improvement. This paper presents a new methodology to compute the criticality of variables in target software applications. Instead of resorting to time consuming fault injection experiments, the proposed solution is based on the run-time analysis of the variables' behavior logged during the execution of the target application under different workloads.*

## 1. Introduction

The use of computer-based systems pervades all areas of our lives from common house appliances, such as microwave ovens and washing machines, to complex applications like aircrafts, trains and medical control systems. In many situations a very large number of new and powerful digital systems play a key role in critical tasks that require human safety and data security.

Devices miniaturization, increasing clock frequencies and the introduction of microprocessors into electrically active environments increase the incidence of transient errors, consequently decreasing the dependability of digital systems [1]. In this scenario, high reliability becomes mandatory to guarantee the required level of dependability. Both NASA and IBM have highlighted that the high miniaturization and high working frequency that represent nowadays circuits working conditions, cause them to be extremely sensitive to the effects of ionizing radiations and noise sources. One of the most probable

consequences of these disturbs is the Single Event Upset (SEU)[2][3] which consists in the change of the content of a single bit of a memory element.

Classical approaches for dependable digital systems development rely on hardware redundancy. Although they are effective in protecting against transient faults, they are usually expensive. To lower costs, software redundancy techniques can be exploited. These approaches are usually referred to as Software Implemented Hardware Fault Tolerance (SIHFT) [4][5].

Different SIHFT techniques have been proposed to address different types of hardware error sources [6][7]. The basic approach consists in improving the Built-In Error Detection Mechanisms of the system (exceptions, memory protection, etc.) by a set of carefully chosen software error detection mechanisms [7]. These techniques include Algorithm Based Fault Tolerance (ABFT) [8], Assertions, and Variable Duplication [9-15].

The present paper focuses on variable duplication techniques which proved to be flexible, general and easy to implement. The basic concept is the duplication of the variables in a program and the insertion of consistency checks before each variable read operation.

In some cases, safety-critical applications have strict constraints in terms of memory occupation and system performances. If on one hand the duplication of the whole set of variables and the introduction of a consistency check before every read operation represent the optimum choice from the fault tolerance point of view, on the other hand the resulting overhead can be unacceptable.

We consider faults appearing in the data memory of each variable only. With this assumption, the duplication of the whole set of variables guarantees 100% of faults covered by the software redundancy technique. On the other side, the duplication of a lower percentage of variables allows

trading-off the fault coverage with the CPU time and memory occupation overhead. One of the problems in this context is how to select the most critical variables to be duplicated out of the whole variable set of the application.

A previously proposed technique to measure the criticality of the variables in a program can be found in [9]. Variable Criticality is intended to be an estimation of the probability to get wrong program results when the variable itself is corrupted. It is strictly related to the harmful situation in which the application ends producing incorrect results while the application gives the impression to terminate correctly. This type of malfunction is called Fail-Silent Violation [16][17]. Researches have shown the relationship between this type of fails and the appearance of a fault in memory locations [2][3].

In [9] the authors use empirical approaches to estimate the criticality of the program variables. These techniques are based on the researcher's know-how and lack of a formal model. Another possibility is to use fault injections techniques, which are very accurate but extremely time consuming. In our approach we will use fault injection only for the initial formalization of a model that will allow us to compute the variable criticality for the variables of any target application.

The paper is organized as follow: Section 2 describes the formal model whereas Section 3 reports experimental results performed on a set of benchmarks. Section 4 shows data about the computation time and finally Section 5 discusses limitation and future improvements whereas Section 6 summarizes the main contributions of the work and concludes the paper.

## 2. Model

The goal of the present work is the definition of an analytical model able to compute the criticality of a variable in a program. Using this model, designers of dependable systems can reduce the overhead introduced by software fault tolerance approaches based on data redundancy (i.e. variable duplication) by identifying critical data only.

What we define is a *Criticality Function*  $CF(v)$  where  $v \in \hat{I}$  {set of variables of the program}. As previously explained the criticality of a variable is intended to be an estimation of the probability to get wrong program results when the variable itself is corrupted, i.e., to have a *Fail Silent Violation* (See Section 1) due to the corruption of the variable; in other words the criticality for each variable is a measure of how an application is sensitive to errors injected in its own memory location.

The  $CF(v)$  is a global value related to a program execution and it represents the average criticality. At each moment it is possible to define an *Instantaneous Criticality Function*  $ICF(v,t)$  where  $v \in \hat{I}$  {set of variables of the program}, representing the criticality of a variable at a given time  $t$  of the program execution. The first step of the model definition is the identification of the  $ICF(v,t)$ .

We define *Life Time* ( $T_{life}$ ) of a variable the time period in which the variable is allocated in memory. During the  $T_{life}$  of a variable, it is possible to identify different events:

- *Creation (C)*: it represents the declaration of the variable in the program. This event causes the variable to be allocated in memory;
- *Write (W)*: it represents an assignment statement in the program;
- *Read (R)*: it represents any instruction involving the use of the variable content;
- *Last Read (LR)*: it represent the last time the variable is used. From this moment on, the variable is still present in memory but its value is no longer used.
- *Death (D)*: the variable is removed from the memory.

Each event modifies the value of the instantaneous criticality. The ICF for a given variable is defined only during the  $T_{life}$ . When a *Creation* event occurs, a memory portion is allocated in order to store the variable's value. At this moment, the ICF is equal to 0 since the variable exists but its content has no meaning for the program and an error in the variable cannot affect its behavior. A *Write* event changes the ICF value. At the *Write* time the content of the variable starts to have a meaning. We define  $K$  the ICF value at *Write* time. It is intuitive that after a *Write* event the ICF value remains constant, since the probability of an error in the variable is constant in time. At each *Read* event, the value of the variable is propagated to others variables. This event can modify the criticality since a corrupted value can be propagated to others variables. This criticality modification can be expressed with an increment  $\beta$  of the ICF. When a *Last Read* event occurs the variable content is no longer used in the program so its criticality drops to zero. Finally, when the *Death* event occurs, the ICF stops to be defined.

Analytically we can define for each instant  $t$  and for each variable  $i$ :

$$ICF_i(v,t,K,b) = \begin{cases} 0 & \text{C} \\ K & \text{W} \\ b + ICF_{i-1}(v,K,b) & \text{R} \\ b + ICF_{i-1}(v,K,b) & \text{LR} \\ 0 & \text{D} \\ 0 & \text{D} \end{cases}$$

A variable can be created and killed several times during a program execution. Figure 1 shows an example of ICF for a single variable. As shown in Figure 1 the ICF is a step function.

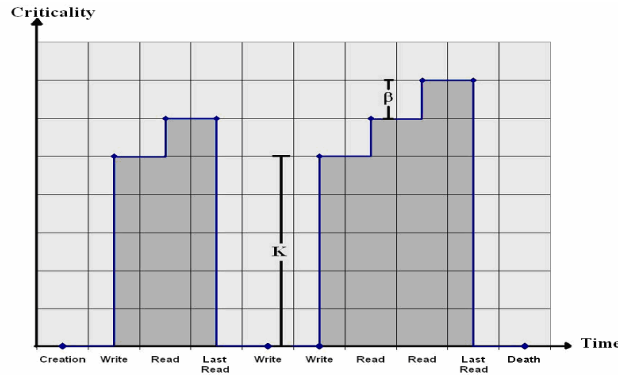


Figure 1: Instantaneous Criticality Function

The ICF has to be defined for each variable in the program. Considering that in a program exist different variable types, some considerations should be made. Pointers in general are more critical than normal variables. To deal with this aspect we introduce a parameter P that multiplies the ICF function in the case of pointers variables.

From the definition of  $ICF(t, v, K, \mathbf{b}, P)$  we can define:

$$CF(v, K, \mathbf{b}, P) = \frac{1}{T_{life}(v)} \int_{T_{life}(v)} ICF(v, t, K, \mathbf{b}, P) \cdot dt$$

The problem to be solved is the correct definition of the parameters K,  $\beta$  and P to have global values not dependent from the application. The definition of K,  $\beta$  and P will be possible through fault injection experiments, but once defined the calculation of the CF and ICF functions will be very fast and precise for any application.

The computation of the model's parameters is performed through a statistical analysis on a golden application. The chosen application is a programmable *Fast Fourier Transfer* (FFT) calculation routine. The correctness of the obtained values of K,  $\beta$  and P will be proven to be general on a set of different benchmarks.

The steps executed to set-up the model (i.e., to calculate the parameters K,  $\beta$  and P) are listed below and explained in details in the next subparagraphs:

1. *Parametric CF computation*: the parametric criticality function  $CF_p(v, K, \mathbf{b}, P)$  is built for the golden application;
2. *Fault Injection Campaign*: an experimental  $CF_e(v)$  is calculated resorting to fault injection experiments;

3. *Parameters computation*: the parameters are computed trying to minimize the difference between the  $CF_p$  and the  $CF_e$ .

## 2.1 Parametric CF Computation

An *ad-hoc* C++ template library has been designed with the main function of logging the behavior of the variables of a target software application. The library records all the events (creation, write, read, last read and death of a variable) occurred during a program execution together with their happening time. The library is based on the concept of C++ templates [18]. Each variable declaration is replaced with an instance of an *ad-hoc* C++ class able to automatically log the previously described events.

The resulting program is functionally equivalent to the original but it generates, at the end of the execution, an event log file useful for our analysis. Starting from the obtained log files a parametric ICF function is calculated according to what defined in Section 2.

The ICF computation is critical in the proposed approach. The accuracy of the results strongly depends on the input stimuli of the target application. To obtain best results during the computation of the ICF the target application should reach all the possible paths. For this reason the golden application has been executed on a set of 16 different input stimuli to be sure to excite all the possible control paths of the application. The log of the different program executions has been chained to obtain an average value for the resulting CF.

As described in Section 2 using the parametric ICF it is possible to obtain the parametric CF as:

$$CF_p(v, K, \mathbf{b}, P) = \frac{1}{T_{life}(v)} \int_{T_{life}(v)} ICF_p(v, t, K, \mathbf{b}, P) \cdot dt$$

## 2.2 Fault Injection Campaign

As mentioned in the introduction, the criticality of a variable is strictly related to the number of *Fail Silent Violations* occurring in a program. Using a fault injection tool able to measure the number of this type of errors it is possible to obtain an experimental criticality function  $CF_e(v)$

A fault injector tool developed in Politecnico di Torino has been used for this purpose. The fault injector is able to deal with the SEU (Single Event Upset) fault model, which consists of a bit flip in a memory location/variable of the program.

Each variable in the golden application has been injected 1000 times. The fault injection campaign allows the classification of the program behavior into three categories:

- *Fail silent violation*: the application ends producing incorrect results;
- *Crash*: the program does not finish;
- *No Effect*: the program produces the correct results.

As mentioned before we consider the number of FSV as the experimental criticality.

### 2.3 Parameters computation

The computation of the parameters  $K$ ,  $\mathbf{b}$ ,  $P$  is performed in order to minimize the difference between  $CF_p$  and the  $CF_e$ . We define a function  $h(\mathbf{b}, K, P)$  as the sum of all the squares deviation  $(CF_e - CF_p)^2$ .

$$h(\mathbf{b}, K, P) = \sum_{v=0}^n (CF_e(v) - CF_p(v, K, \mathbf{b}, P))^2$$

The problem is now the definition of  $K$ ,  $\mathbf{b}$ , and  $P$  able to minimize the function  $h(\mathbf{b}, K, P)$ . This problem can be expressed in a formal way as follow:

$$\begin{cases} \frac{\partial h(\mathbf{b}, K, P)}{\partial \mathbf{b}} = 0 \\ \frac{\partial h(\mathbf{b}, K, P)}{\partial K} = 0 \\ \frac{\partial h(\mathbf{b}, K, P)}{\partial P} = 0 \\ K \geq 0 \\ P \geq 0 \end{cases}$$

The two constraints are introduced to avoid negative values for the resulting criticality.

The problem has been solved using Derive™ software by Texas Instruments [19] and the resulting values are listed below:

$$\begin{aligned} \hat{\mathbf{b}} &= 4.46 * 10^{-5} \\ \hat{K} &= 89.14 \\ \hat{P} &= 1.14 \end{aligned}$$

The values appear to be reasonable. The  $P$  parameter confirms our hypothesis that an error into a pointer is more catastrophic than an error into others variables. The value of  $K$  shows that the main contribution to the criticality is given by the time elapsed between *write* and *last-read* operations whereas the low value for  $\mathbf{b}$  means that the criticality is only slightly influenced by the number of read operations to the variable. This is an interesting result since it is in contrast with the empirical approach proposed in [9].

Figure 2 shows a graph containing the  $CF_e(v)$  and the  $CF_p(v, K^*, \mathbf{b}^*, P^*)$  ordered by variable criticality. As previously explained,  $CF_e(v)$  is obtained by performing fault injection campaign on the target application whereas  $CF_p(v, K^*, \mathbf{b}^*, P^*)$  is analytically computed using the proposed model. As expected the two values are similar.

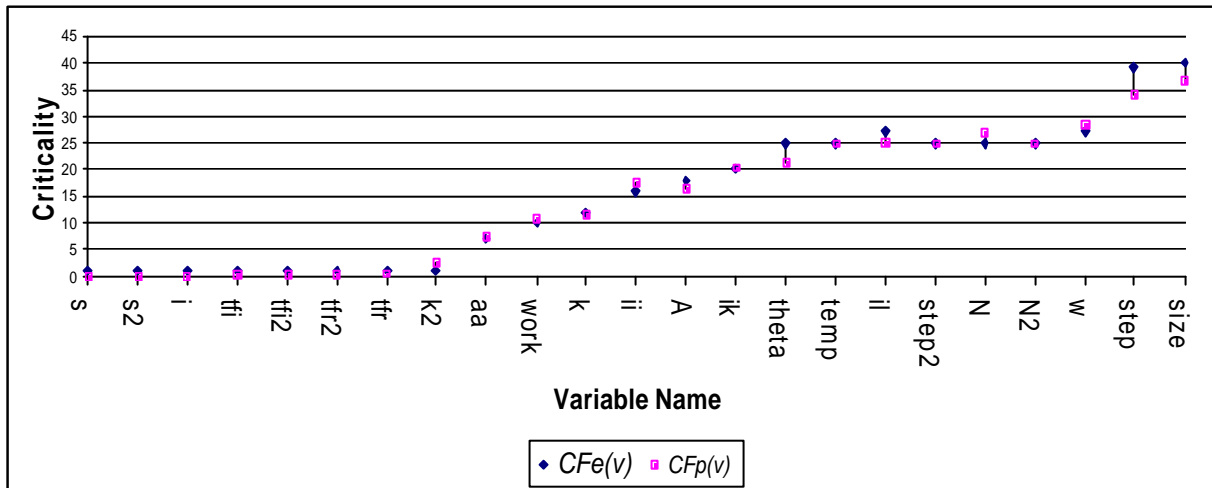


Figure 2:  $CF_e(v)$  and the  $CF_p(v, K^*, \mathbf{b}^*, P^*)$  for the FFT application

What we have to demonstrate in the next section is that the same parameters are valid if used to compute the  $CF_p(v, K, \mathbf{b}, P^*)$  for a different application, so that the criticality of a variable can be obtained without resorting to fault injection experiments.

### 3. Experimental Results

A collection of experiments has been set up to prove the effectiveness of the approach and to show that the values of  $\mathbf{b}$ ,  $K$  and  $P$  are application independent.

Four programs have been used to validate the model: the Dhrystone benchmark, two custom int/float performance benchmarks and the jpeg-2000 compression algorithm. The performance benchmarks have been designed to stress the model to a maximum extent: the former (Performance Benchmark 1) is composed of variables having very different criticalities, i.e., some variables are read much more than others; the second (Speed Benchmark 2) is composed of variables with the same access rate, i.e., all the variables are read and written with the same frequency.

For each benchmark we obtained the experimental criticality of the variables using fault injection. Each variable has been injected one thousand times to obtain its criticality value. In a second step the parametric criticality  $CF_p(K, \mathbf{b}, P)$  has been computed using the proposed model where the values of  $K$ ,  $\mathbf{b}$  and  $P$  are the ones shown in Section 3.

The following figures show, for each benchmark and for each variable, the two criticalities and the standard deviations between the two values. It is evident that, the values of  $CF_p$  are very close to the values of  $CF_e$  obtained by fault injection. Since the two performance benchmarks have been designed to stress the model the standard deviations are higher w.r.t. a general application like the Dhrystone but anyway their values are acceptable. These results allow us to state that the values of  $K$ ,  $\mathbf{b}$  and  $P$  obtained in Section 3 are general enough to be reused with others applications. Obviously to be more confident in the obtained results the model should be applied to a more significant number of test cases in order to obtain statistical results and to refine the model's parameters.

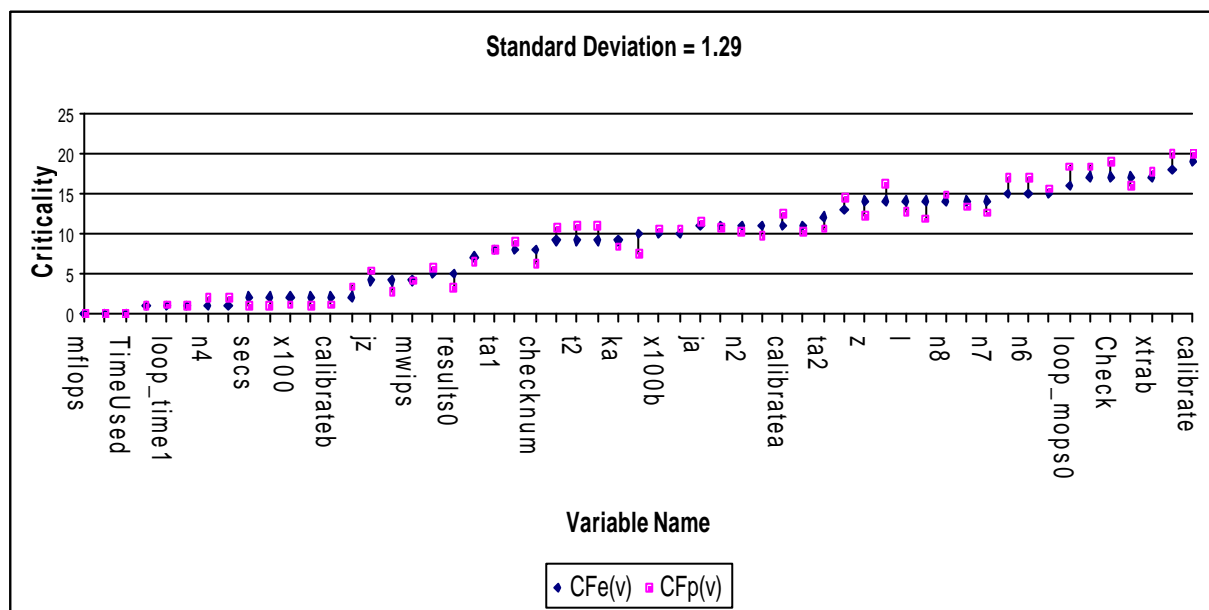


Figure 3:  $CF_e(v)$  and the  $CF_p(v, K, \mathbf{b}, P^*)$  for the Dhrystone

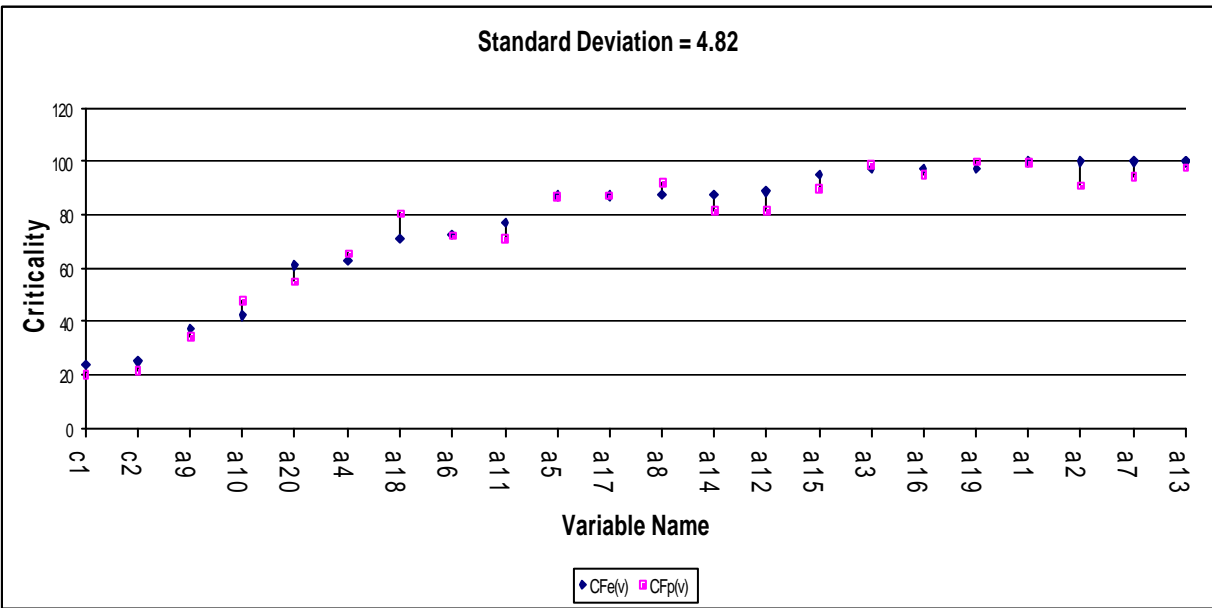


Figure 4: CFe(v) and the CFP(v,K\*,b\*,P\*) for the Performance Benchmark 1

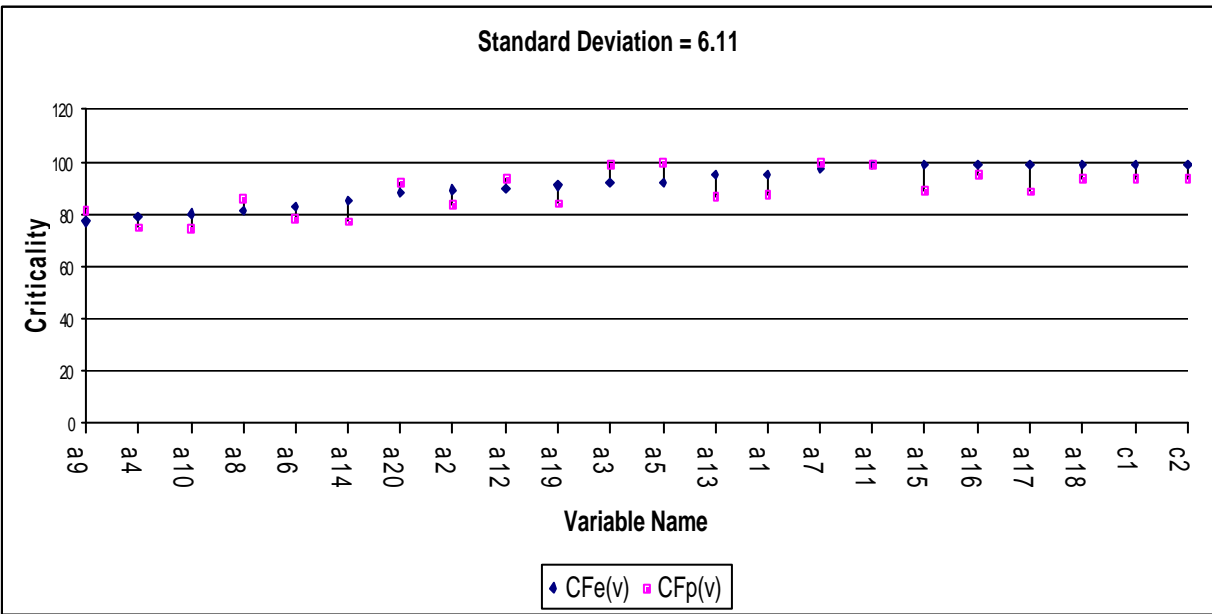


Figure 5: CFe(v) and the CFP(v,K\*,b\*,P\*) for the Performance Benchmark 2

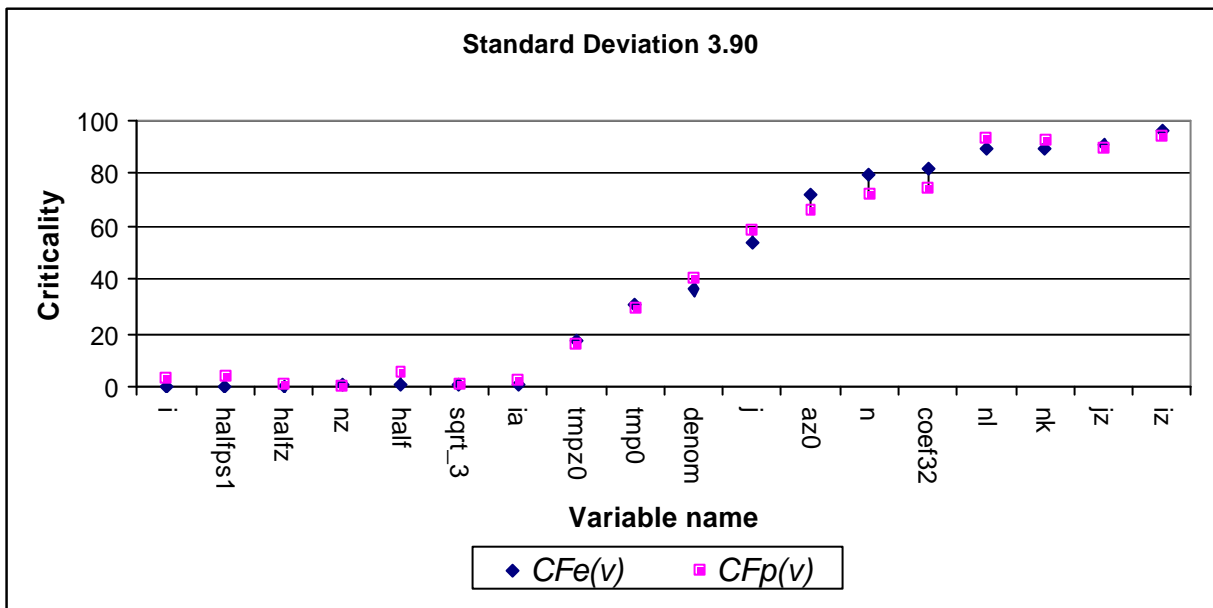


Figure 6:  $C_{Fe}(v)$  and the  $C_{Fp}(v, K^*, b^*, P^*)$  for the JPEG-2000 compression

To underline the accuracy in evaluating variable criticality we compare the results obtained by our experiments with the ones obtained by the empirical method presented in [9] (ReCCo). Figure 7 shows, for the golden application, the FSV reduction related to the percentage of duplicated

variables. It is clear that with our new methodology it is possible to reach higher dependability levels using a lower amount of redundancy (i.e., number of duplicated variables).

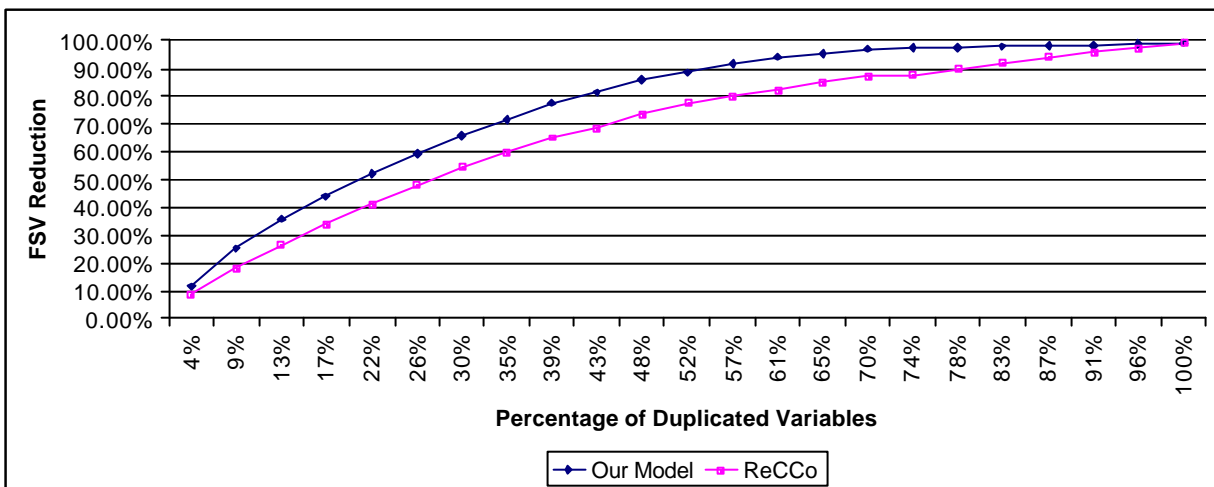


Figure 7: Reduction of Fail-Silent Violations (FSV) via Variable Duplication

#### 4. Timings

One of the goals of our approach is to reduce the time needed to estimate the criticality of the variables. Whereas fault injection experiments are always very time consuming, the proposed approach is based on few executions of the target application only. The number of

executions depends on the set of inputs needed to excite all the control paths of the application under test.

To demonstrate why it is more convenient to compute the criticality using our methodology instead of fault injection, we compared the time required for the four presented injection campaigns with the time required to compute the criticality using the proposed approach.

Table 1 shows the time needed to compute the  $CF_e$  (using fault injection campaign) and the related time needed to

obtain the  $CF_p$  (using the proposed model). It can be seen in all the test cases reduction ranges from 94% to 98%.

	Variables #	Injections #	Injection Time (s)	CFp Time (s)	Time reduction
<b>FFT</b>	23	23000	14950	700	96%
<b>Dhrystone</b>	56	56000	117600	2250	98%
<b>Benchmark1</b>	22	22000	11000	600	94%
<b>Benchmark2</b>	22	22000	11000	600	94%
<b>JPEG-2000</b>	18	18000	42000	1300	97%

Table 1: CF<sub>e</sub> and CF<sub>p</sub> timings

## 5. Limitation and Future Work

One of the main limitations in the proposed approach is in the computation of the ICF. This critical task requires the execution of the target application under a set of input stimuli able to excite all the possible control flow paths.

A non-complete set of input stimuli can lead to an erroneous value of the criticality. Until now, the task of generating the input stimuli has been performed by hand. This approach is feasible for small application only. To extend the approach to larger applications this task has to be automated.

Similar problems are common in the field of Software Testing. Future works can investigate the use of formal approach for the generation of the input stimuli of the application.

## 6. Conclusion

This paper proposed a formal methodology to compute the criticality of the set of variable of a program. In contrast with the previously proposed techniques, which are based on empirical approaches and on fault injections experiments, the proposed methodology uses fault injection only for the initial formalization of a model that allows computing the criticality for the variables of any target application. The main advantages of the model are formalization, accuracy of the results and low computation time. The methodology has been applied on a set of different benchmarks with results always comparable to the ones obtained using fault injection. The experimental results show also that the computed criticality allows higher dependability level with lower percentage of duplicated variables with respect to the previously proposed approaches.

## 7. References

[1] F. Faccio, C. Detcheverry, M. Huhtinen CERN, Geneva, Switzerland, "First evaluation of the Single Event Upset (SEU) risk for electronics in the CMS

*Experiment*"; CMS NOTE 1998/054 CERN, Geneva, Switzerland.  
 [2] <http://tvdg10.phy.bnl.gov/seutest.html>  
 [3] <http://www.research.ibm.com/journal/rd/ziegl/Ziegler.html>  
 [4] P.P. Shirvani, N. Oh, E.J. McCluskey, D.L. Wood, M.N. Lovellette, K.S. Wood, "Software-Implemented Hardware Fault Tolerance Experiments: COTS in Space" International Conference on Dependable Systems and Networks (FTCS-30 and DCCA-8), New York (NY), 2000, Page(s) B56-57.  
 [5] P.P. Shirvani, N.R. Saxena, E.J. McCluskey, "Software-implemented EDAC protection against SEUs", IEEE Transactions on Reliability, vol. 49 Issue: 3, Sep 2000, Page(s): 273 -284.  
 [6] D. Todd Smith, T. A DeLong, B. W. Johnson, J. A. Profeta III, "An Algorithm Based Fault Tolerant Technique for Safety Critical Applications", Reliability and Maintainability Symposium, Philadelphia, 1997.  
 [7] M. Zenha Relá, H. Madeira, J. G. Silva, "Experimental Evaluation of the Fail-Silent Behavior in Programs with Consistency Checks", 26th International Symposium on Fault-Tolerant Computing (FTCS-26), Sendaj (J), 1996, Page(s). 394-403.  
 [8] K. H. Huang, J. A. Abraham, "Algorithm-Based Fault Tolerance for Matrix Operations", IEEE Transaction on Computers, vol. 33, 1984, Page(s) 518-528-  
 [9] A. Benso, S. Chiusano, P. Prinetto. L. Tagliaferri, "C/C++ source-to-source compiler for dependable applications", International Conference on Dependable Systems and Networks, (FTCS-30 and DCCA-8), New York (NY), 2000, Page(s): 71 -78.  
 [10] A. Benso, S. Chiusano, P. Prinetto, "A software development kit for dependable applications in embedded systems", International Test Conference (ITC 2000), Atlantic City, NJ, 2000, Page(s): 170 -178.  
 [11] A. Benso, S. Di Carlo, G. Di Natale, L. Tagliaferri, P. Prinetto, "Validation of a software dependability

- tool via fault injection experiments*”, Seventh International On-Line Testing Workshop (IOLTW 2001), Taormina (IT), 2001, Page(s): 3 -8.
- [12] M. Rebaudengo, M. Sonza Reorda, M. Violante, P. Cheynet, B. Nicolescu, R. Velazco, “*System safety through automatic high-level code transformations: an experimental evaluation*”, Design Automation and Test in Europe (DATE 2001), Munich (DE), 2001, Page(s): 297 -301.
- [13] B. Nicolescu, R. Velazco, M. Sonza-Reorda, M. Rebaudengo, M. Violante, “*A software fault tolerance method for safety-critical systems: effectiveness and drawbacks*”, 15th Symposium on Integrated Circuits and Systems Design, Porto Alegre (Brazil), 2002, Page(s): 101 -106.
- [14] M. Rebaudengo, M. Reorda, M.S. Violante, M. Torchiano, “*A source-to-source compiler for generating dependable software*”, First IEEE International Workshop on Source Code Analysis and Manipulation (SCAM 2001), Florence (Italy), 2001, Page(s): 33 -42.
- [15] M. Rebaudengo, M. Sonza Reorda, M. Torchiano, M. Violante, “*An experimental evaluation of the effectiveness of automatic rule-based transformations for safety-critical applications*”, IEEE International Symposium on Defect and Fault Tolerance in VLSI Systems (DFT 2000), Yamanashi (Japan), 2000, Page(s): 257 -265.
- [16] A. M. Amendola, A. Benso, F. Corno, L. Impagliazzo, P. Marmo, P. Prinetto, M. Rebaudengo, M. Sonza Reorda, “*Fault Behavior Observation of a Microprocessor System through a VHDL Simulation-Based Fault Injection Experiment*”, EURO-VHDL'96, 1996, Geneva (CH), Page(s): 536-541.
- [17] J. G. Silva, J. Carreira, H. Madeira, D. Costa, F. Moreira, “*Experimental Assessment of Parallel Systems*”, 26th International Symposium on Fault-Tolerant Computing (FTCS-26), Sendaj (J), 1996, Page(s). 415-424
- [18] The C++ Programming Language (Third Edition and Special Edition) Addison-Wesley, ISBN 0-201-88954-4 and 0-201-70073-5.
- [19] <http://education.ti.com/>