EXFI: a low cost Fault Injection System for embedded Microprocessor-based Boards

Authors: Benso A., Prinetto P., Rebaudengo M., Sonza Rreorda M.,

DOI: 10.1145/296333.296351

EXFI: a low cost Fault Injection System for embedded Microprocessor-based Boards

A. Benso
Politecnico di Torino
Dip. Automatica e Informatica
corso Duca degli Abruzzi 24
I-10129 Torino, Italy
Tel. +39 11 564 7055
Fax +39 11 564 7099
E-mail sonza@polito.it

Alfredo BENSO was born in Torino, Italy, in 1970. He received the M.S. degree in computer science engineering from the Politecnico di Torino, Italy, in 1995 and he is currently a Ph.D. student at the Politecnico di Torino. His research interests include Design for Testability Techniques and Dependability analysis of computer-based systems.

P. Prinetto

Paolo Prinetto was born in Gassino Torinese, Italy on March 17, 1953.
He received his M.S. in Electronic Engineering in 1976 from the Politecnico di Torino, Italy.
Since 1990 he is full professor of Computer Science at the Dipartimento di Automatica e Informatica of the same university.
His research interests cover systems and tools for CAD for VLSI, with particular emphasis on testing, automated testable synthesis, hardware description languages, formal verification of correctness of digital designs, system level description and validation.
Since 1994, he is the chairman of ETTTC: the European Group of the IEEE-Computer Society Test Technology Technical Committee (TTTC). In 1996 he became *IEEE Computer Society Golden Core Member*.

Politecnico di Torino
Dip. Automatica e Informatica
corso Duca degli Abruzzi 24
I-10129 Torino, Italy
Tel. +39 11 564 7055
Fax +39 11 564 7099
E-mail sonza@polito.it

M. Rebaudengo
Politecnico di Torino
Dip. Automatica e Informatica
corso Duca degli Abruzzi 24
I-10129 Torino, Italy
Tel. +39 11 564 7055
Fax +39 11 564 7099
E-mail sonza@polito.it

Maurizio REBAUDENGO received the M.S. degree in Electronics in 1991, and the Ph.D. degree in Computer Science in 1995, both from the Politecnico di Torino, Torino, Italy. In 1997 he became a researcher at the Department of Computer Science and Automation of the same Institution. His research interests include Automatic Test

Pattern Generation, parallel and distributed algorithms for Electronic CAD, and dependability analysis of computer-based systems.

M. Sonza Reorda
Politecnico di Torino
Dip. Automatica e Informatica
corso Duca degli Abruzzi 24
I-10129 Torino, Italy
Tel. +39 11 564 7055
Fax +39 11 564 7099
E-mail sonza@polito.it
Matteo SONZA REORDA received the M.S. degree in Electronics in 1986, and the Ph.D. Degree in Computer Science in 1990, both from the Politecnico di Torino, Torino, Italy. He currently is an Assistant Professor at the Dept. of Computer Science and Automation of the same Institution. His research interests include testing of digital systems, design verification methodologies, and fault injection techniques.

Evaluating the faulty behavior of low-cost embedded microprocessor-based boards is an increasingly important issue, due to their adoption in many safety critical systems. The architecture of a complete Fault Injection environment is proposed, integrating a module for generating a collapsed list of faults, and another for performing their injection and gathering the results. To address this issue, the paper describes a software-implemented Fault Injection approach based on the Trace Exception Mode available in most microprocessors. The authors describe EXFI, a prototypical system implementing the approach, and provide data about some sample benchmark applications. The main advantages of EXFI are the low cost, the good portability, and the high efficiency.

1 INTRODUCTION

Our society is facing with an increasing dependence on computing systems, even in areas (e.g., air and railway traffic control, nuclear plant control, aircraft and car control) where a failure can be critical for the safety of human beings. As a consequence, the past years have seen a growing interest in methods for studying the behavior of computer-based systems when faults occur, and several approaches have been proposed to evaluate the dependability properties of a computer-based system.

In many cases, Fault Injection [1] emerged as a viable solution, and has been deeply investigated by both academia and industry. Different techniques have been proposed and some of them practically experimented. One of the current challenges in the area is how to adapt these techniques to assess the hardware and software fault detection capabilities of high-volume, low-price microprocessor- (and microcontroller-) based safety critical products (e.g., those used in the automotive sector).

The goal of this paper is to present a software-implemented Fault Injection system, named EXFI (<u>Ex</u>ception-based <u>F</u>ault <u>I</u>njector), suited to be used in embedded microprocessor-based boards.

The kernel of the EXFI system is based on the *Trace Exception Mode* available in most microprocessors. During the Fault Injection experiment, the trace exception handler routine is in charge of computing the Fault Injection time, executing the injection of the fault, and triggering a possible time-out condition. The tool is able to inject single bit-flip transient faults both in the memory image of the process (data and code) and in the user registers of the processor. The approach can be easily extended to support different fault models, such as permanent stuck-at, coupling, temporal and spatial multiple bit-flip, etc.

A *Fault List Manager* is also included in the system to generate a collapsed list of faults to be injected. This module exploit the collapsing rules defined in [2], practically demonstrating how they can be implemented and how effective they really are.

A case study is presented in which a Motorola M68KIDP board [3] based on a M68040 microprocessor is considered; a prototypical version of EXFI has been implemented, and some sample application programs are considered.

The main characteristics of EXFI are the low cost (it does not require any hardware device), the high speed (which allows a higher number of faults to be considered), the low requirements in terms of features provided by the Operating System, the flexibility (it supports different fault types), and the high portability (it can be easily migrated to address different target systems).

The paper is organized as follows: after discussing some related research in Section 2, Section 3 states the adopted assumptions. Section 4 describes the Fault Injection environment, and Section 5 reports some experimental results. Some conclusions are eventually drawn in Section 6.

2 RELATED RESEARCH

The many different approaches proposed to implement Fault Injection (a detailed discussion can be found in [1]) can be categorized in three main groups:

- *Hardware-implemented* Fault Injection: errors are emulated by changing the state of the system, either forcing faulty values on the pins of the chips [4], or injecting faults inside the chips through heavy-ion radiation [5]. These techniques require special and often expensive hardware, and do not allow injecting faults in every possible location.
- *Software-implemented* Fault Injection: faults are injected under software control. The main advantages of software approach are the low complexity, the high speed of the experiment, and the higher flexibility in the set of injectable faults. The main disadvantage is the high degree of intrusiveness in the target system.
- *Simulation-based* Fault Injection: faults are injected into a model of the system and its behavior is analyzed through simulation [6], [7]. This approach allows the

maximum flexibility in the type of faults that can be injected. Unfortunately, it usually involves a high effort for developing the system descriptions, and the experiments are mostly very time consuming.

To have a fast and low cost solution to dependability evaluation problems, the software-implemented approach can be the most effective one. In fact, when simple boards have to be analyzed, hardware fault injectors are too cumbersome and too expensive. On the other hand, simulation-based fault injectors require the development of highly complex descriptions and are too time consuming and ineffective.

Several different solutions for software-implemented Fault Injection have been proposed:

- *FERRARI* [8] uses software traps and trap handling routines to inject faults in CPU, memory and bus. Experiments are conducted on a Sun SparcStation, and the target system adopts Operating system calls, such as the UNIX *ptrace*, to corrupt the process memory image.
- *Xception* [9] uses a processor's built-in hardware exception mechanism to trigger Fault Injection. Faults are triggered based on access to specific addresses. Xception has been implemented on a system based on a PowerPC 601 processor.

By exploiting the microprocessor trace mode, EXFI does not require any change in the source code of the target software: with respect to FERRARI, our approach is oriented to simple embedded microprocessor systems, rather than to complex workstation-based ones. As a consequence, EXFI exploits the basic target microprocessor facilities, and the system is not supposed to provide any Operating system calls, such as the ones used in FERRARI. Moreover, EXFI does not insert software traps or Fault Injection routines in the target software, thus greatly limiting its intrusiveness. When compared with Xception, EXFI does not need any specific debugging features, as the ones exploited by the Xception tool for the PowerPC processor.

Moreover, it is worth noting that the paper describes a Fault Injection *environment*, providing the user with a full range of well-integrated features, ranging from Fault List Generation and Collapsing, to an effective Fault Injection technique, and to a simple way for analyzing the faulty system behavior. The reported experience on some sample benchmark applications provides information about the usability of the environment.

As a result, the EXFI approach is well suited for simple and low-cost systems, where the operating system support is not available, and the effort for setting up the Fault Injection experiments has to be very small.

3 ASSUMPTIONS

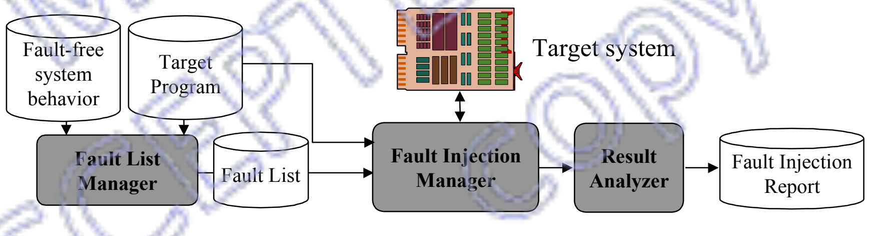
The adopted fault model is the transient fault. This model is frequently used in Fault Injection tools [8] [7] since it is very similar to the faults occurring in real systems [10]. The fault type adopted in the preliminary version of the tool is the single bit-flip, also known as Single Upset Event (SEU), but the approach can be extended to other kinds of fault models. The Fault Injection time is expressed in terms of number of instructions executed since the beginning of the application execution. Faults can be injected in any memory location or register accessible through an Assembly instruction.

Our technique is ideally suited to systems whose behavior, when a sequence of input stimuli is applied, can be deterministically computed and easily reproduced. To detect a target system faulty behavior we rely on the built-in Error Detection Mechanisms (EDMs), as system exceptions or software checks.

In the present version we do not address the issue of checking the system behavior from the time point of view: the extension to real-time systems composed of several interacting modules is currently under study.

4 FAULT INJECTION SYSTEM

As illustrated in Figure 1, the EXFI Fault Injection system can be divided in three modules. The Fault List Manager (FLM) generates the fault list to be injected into the system, the Fault Injection Manager (FIM) injects the faults into the system, and the Result Analyzer collects the results and produces a report concerning the whole Fault Injection experiment.



Fi‸ure 1: The EXFI Fault Injection System.

4.1 FAULT LIST MANAGER

The Fault List Manager (FLM) generates the list of faults which are then injected in the target system by the Fault Injection Manager. Since the fault list size is a crucial parameter that directly affects the time required to perform the Fault Injection experiment, special care has been devoted to devise techniques, able to reduce the size of the Fault List, without reducing the meaningfulness of the Fault Injection results.

The architecture of the EXFI FLM is based on two modules: the Fault List Generator and the Fault List Collapser.

The *Fault List Generator* generates a Fault List according to some input constraints (e.g., number of faults, boundaries of the used memory area, statistical distribution of faults, etc.). The *Fault List Collapser* implements the rules introduced in [2] to process and possibly collapse the Fault List generated by the previous module. These rules aim at avoiding the injection of those faults whose behavior can be foreseen a priori, without affecting the accuracy of the results gathered through the Fault Injection experiments. The validity of the collapsing rules is bounded to the considered Fault Injection environment, and to the set of input data stimuli the target system is going to receive.
A fault can be removed from the fault list when it fits in one of the following classes:

- the fault is guaranteed to trigger an Error Detection Mechanism, e.g., because it affects the operative code of an instruction and changes it into an *illegal operative* code;
- it is guaranteed not to have any effect, e.g., because it affects the code of an instruction after the very last time the instruction is executed;
- it is equivalent to another fault already existing in the fault list, e.g., the two faults flip the same bit in the code of the same instruction during the period between two subsequent executions of the same instruction; the two faults are equivalent since they behave in the same way, and can thus be collapsed to a single fault.

To collapse the fault list, the Fault List Collapser exploits the information collected during a preliminary *golden-run* experiment, in which the behavior of the Fault-Free System is observed and recorded. During this experiment, a modified trace procedure is used to record the sequence of executed instructions, and a post-processing phase elaborates the recorded information to assess the sequence of accesses performed to registers and memory variables.

The Fault List Collapser also generates a report containing the information about the collapsed faults. This report is then used by the Result Analyzer to provide statistical results concerning the original Fault List.

4.2 FAULT INJECTION MANAGER

The Fault Injection Manager (FIM) is the most crucial part in the whole Fault Injection System. It is up to the FIM to activate the execution of the target application once for each fault in the list generated by the Fault List Manager, to inject the fault at the required time and location, and to observe the system behavior after the Fault Injection. The sequence of operations executed by the Fault Injection Manager is outlined in Figure 2 and explained in the following section.

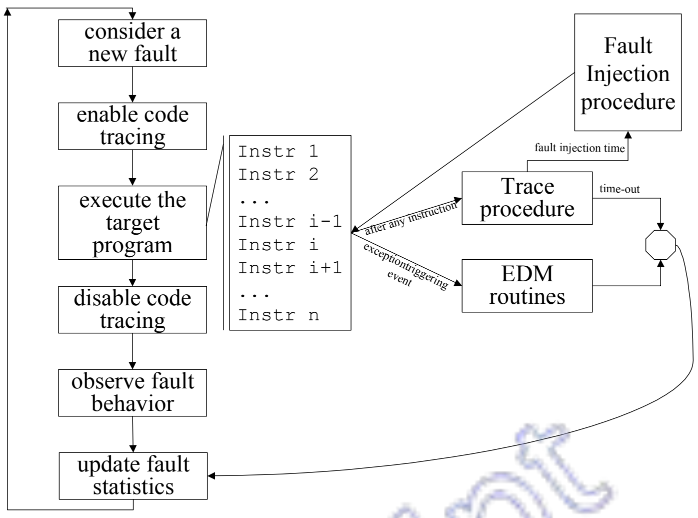


Figure 2: Fault Injection Manager operation flow.

The main issues to be faced when devising and implementing an effective Fault Injection Manager are:

- Identification of the fault injection time and Fault Injection; the target application execution must be continuously monitored and, when the fault injection time is reached, the fault injection according to the fault type (e.g., single bit-flip) and location specified in the Fault List must be performed.
- Fault Effects Observation: the system behavior after fault injection must be observed, and differences with respect to the fault-free system behavior identified. This requires the implementation of some time-out mechanism for the identification of faults forcing the system in endless loops.
- Recovery from fault effects: the FIM should be able to recover from the effects generated by the injection of any fault; this requires that the FIM maintains the system control even in the likely event of a hardware exception being triggered. Moreover, the FIM should ensure that, for all the faults, the target application be run in the same fault-free initial environment, therefore avoiding that the effects of any fault (e.g., corrupted bit in data and code memory sections) be still present in the environment where the following experiment is run.

The above tasks have to be accomplished while the target application is running and with a minimum intrusiveness with respect to its behavior.

4.2.1 PROPOSED SOLUTION

The following paragraphs describe the different modules that compose the overall FIM module.

Experiment Initialization

This module initializes the system and prepares the environment for the Fault Injection into the target application program.

It first makes a *golden copy* of the target and FIM program code and input data into a safe part of the system memory (i.e., one that can not be modified by fault effects). This can often be obtained by exploiting the memory protection mechanism provided by the Memory Management Unit integrated in most microprocessors. In this way, the FIM can start each new Fault Injection experiment using a known fault-free copy of data and code.

The second task of this module is to create a new Exception Vector Table in order to replace the original exception processing routines with the new ones, which provide the Fault Injection and system monitoring capabilities, as described in the following.

Initialization of the environment for the injection of the single fault

The first task of this module is to restore the golden copy of the target application program to the memory area, where the program is going to be executed during the experiment. This operation is necessary to start a new experiment with a fault-free version of the target code and data.

The second task of this module is to initialize some variables (e.g., the ones storing the information about the fault to be injected).

Finally, the module enables the code tracing and jump to the first instruction of the target application code.

Trace Procedure

The Trace Procedure performs two main tasks:

- It injects a fault into the system: each time the procedure is executed, a variable that stores the number of executed instructions is incremented. As soon as this value matches the *injection time* of the fault that has to be injected, the procedure performs its injection.
- It monitors the instruction counter; if its value exceeds a user-defined limit the experiment is terminated and the fault is classified among those producing a *time-out*.

Exception Routines

In most microprocessors, an exception (or internal interrupt) is activated when some incorrect operation is performed or simply attempted. In such a case, a procedure is automatically activated. This mechanism can be exploited to implement an Error Detection Mechanism able to detect all faults triggering an exception during the system activity. The exception routines is suitably modified and performs two tasks:

- It updates the data structure containing the information about the faults behavior
- It returns the program execution to the main FIM loop. To perform this task, the procedure modifies the return address stored in the stack so that the execution of the return assembly instruction returns the execution control to the Experiment Control Loop instead of to the instruction that triggered the exception. In this way, no matter the type of exception triggered by the fault, it is possible to 'recover' the error and start the injection of a new fault.

Target Application Result Check Routine

A computer-based system is said to be *Fail-Silent* if it outputs only correct results, i.e., if the system does not output incorrect results even if they are possibly generated internally as a consequence of a fault [11]. Many researchers [12] have shown that, in computer-based systems, a high percentage of faults cause a *Fail-Silent Violation* behavior, e.g., the system produces incorrect results while neither the EDMs nor the time-out checks are activated. Therefore, it is necessary that the application programmer provides a procedure (in Figure 2 called *Observe_Fault_Behavior*) able to verify the correctness of the results produced by the target application execution when it terminates without triggering any exception or time-out condition.
Faults are classified according to four main categories:

- *Fail-Silent*: the fault has no effect on the system behavior.
- *Detected by an EDM*: the faulty system behavior triggers the activation of either a software or hardware EDM.
- *Fail-Silent Violation*: the faulty system behavior does not trigger any EDM, and the output results are different from the fault-free ones.
- *Time-out*: this category includes faults triggering the time-out condition. These faults alter the system behavior from a temporal point of view without triggering any EDM.

4.3 RESULT ANALYZER

The *Result Analyzer* processes the system output behavior obtained through Fault Injection experiments and the report on collapsed faults generated by the Fault List Manager. The Result Analyzer produces a report concerning fault coverage information referred to the whole Fault List.

5 EXPERIMENTAL RESULTS

To evaluate the effectiveness of our Fault Injection approach, a case study is described below.
The EXFI environment has been implemented on a commercial M68KIDP Motorola board [3]. This board hosts a M68040 microprocessor with a 25Mhz frequency clock,

2 Mbytes of RAM memory, 2 RS-232 Serial I/O Channels, a Parallel Printer Port, and a bus-compatible Ethernet card.

Some simple programs have been adopted as benchmark target applications:

- *Bubble Sort*: an implementation of the bubble sort algorithm, run on a vector of 10 integer elements;
- *Parser*: a syntactical analyzer for arithmetic expressions written in ASCII format. The program also implements a simple software Error Detection Mechanism, which consists in verifying the correctness of each part of the expression;
- *Matrix*: multiplication of two matrices composed of 10x10 integer values.

For each target program, the original Fault List is composed of 30,000 randomly selected faults located in the code (10,000 faults) and data (10,000 faults) memory area, as well as in the microprocessor registers (10,000 faults). The original Lists of faults located in the code area and in the registers are then collapsed with respect to the given sequence of Input Stimuli. Due to the complexity of the post-processing phase, collapsing of faults in the data memory area is not implemented in the current version of EXFI.

The results of the collapsing phase are reported in Table 1. They show very different collapsing figures depending on the benchmark program and fault location.

In general, the percentage of collapsed faults among those to be injected in the code is quite stable. The amounts of faults activating an EDM mainly depends on the ratio between legal and illegal codes resulting from the microprocessor instruction set definition. On the other side, the collapsing figures for faults in data and registers mainly depend on the kind of application we are considering: in particular, one crucial parameter is the size of the data structures. This parameter strongly affects the percentage of *no-effect* faults, since larger data structures often imply a higher number of faults injected in a variable or register outside the period in which it is used. This is demonstrated by the high percent of faults removed by the Fault Collapser because they do not produce any effect in the Matrix benchmark, which has the largest data structures among the three considered programs. Moreover, the percentage of collapsed faults among those to be injected in the registers also depends on the complexity of the considered application and in the compiler capabilities in exploiting the available registers: in general, an intensive register usage reduces the effectiveness of the fault collapsing rules identifying equivalent and no-effect faults.

	Bubble Sort		*Parser*		*Matrix*	
Location	*Code*	*Reg.*	*Code*	*Reg.*	*Code*	*Reg.*
initial fault list size	10,000	10,000	10,000	10,000	10,000	10,000
Total removed faults	1,575	5,361	1,488	6,386	1,323	7,487
Detected by an EDM	792	127	671	154	788	139
No effect	510	4,595	525	4,125	323	6,317
Fault Equivalence	273	639	292	2,107	212	1,031

Table 1: Fault collapsing figures for faults injected in the code and in the registers.

Based on the Fault Lists generated by the Fault List Manager, the Fault Injection Manager performed the Fault Injection experiments, whose results are reported in Tables 2, and 3.

	Bubble Sort		*Parser*		*Matrix*	
	Code	Data	Code	Data	Code	Data
Fault Category	%	%	%	%	%	%
Fail-Silent	58.10	66.11	63.96	64.34	50.25	16.35
Fail-Silent Violation	24.18	31.20	13.42	10.34	25.15	81.55
Detected by an EDM	15.75	2.20	20.24	24.30	22.62	1.50
Time-out	1.97	0.49	2.38	1.02	1.98	0.60

Table 2: Faults injection report for the faults injected in the code and data area.

	Bubble Sort	*Parser*	*Matrix*
Fault Category	%	%	%
Fail-Silent	70.81	82.73	71.23
Fail-Silent Violation	2.97	2.99	3.18
Detected by an EDM	17.09	13.48	15.24
Time-out	9.12	0.80	10.35

Table 3: Faults injection report for the faults injected in the registers.

The results of Tables 2 to 3 confirm that the behavior of faults injected in the code area is more regular than that of the faults injected in the data area, which highly depends on the characteristics of the considered application. As a further example, the reader should observe the very different percentages of Fail-Silent and Fail-Silent Violation Faults reported for the three benchmarks among those injected in the data area. *Bubble* and *Parser* are control-dominated programs: many variables (e.g., those associated with flags and loop indexes) are used for the execution flow control, and faults injected in them are likely to either trigger an EDM, or be fail-silent. On the other side, *Matrix* is data-dominated, and most variables contain data rather than control information. Faults injected in them are therefore more likely to generate Fail-Silent Violations.

The Result Analyzer collects the results produced by the Fault Injection Manager and takes into account the collapsing information provided by the Fault List Manager. The complete Fault Coverage figures with respect to the initial Fault Lists are reported in Table 4.

	Bubble Sort	*Parser*	*Matrix*
	%	%	%
Fail-Silent	60.62	62.86	32.09
Fail-Silent Violation	26.35	11.18	52.19
Detected by an EDM	11.98	24.40	14.54
Time-out	1.06	1.56	1.19

Table 4: Summary of Faults injection results.

To quantitatively evaluate the time required to perform a Fault Injection experiment, we compared the total time required to perform the Fault Injection of 30,000 faults with the one required to execute 30,000 time the same program with the same input data in normal mode and without injecting any fault. The resulting ratio ranges between 20 and 22 for the considered benchmarks; the differences are mainly due to the different collapsing ratios obtained through the FLM.

6 CONCLUSIONS

In this paper we presented a Software-based Fault Injection environment suitable to be used for fault coverage evaluation on embedded microprocessor-based boards.

Our environment is composed of three main parts: the Fault List Manager to generate and collapse the Fault List, the Fault Injection Manager to perform Fault Injection, and the Result Analyzer to produce output reports.

During the Fault Injection experiments, the target application program is executed in trace mode and faults are injected by a suitably modified exception handler routine. In this way, faults can be injected into any location accessible through an Assembly instruction. Faults are injected without any change in the target application code and with very limited intrusiveness in the system behavior, the only overhead being in terms of an increase in the execution time with respect to a fault-free system.

The approach is quite general and flexible, as it is based on common features supported by most microprocessors. Moreover, it does require neither dedicated hardware, nor any Operating system being present on the board, thus matching well the constraints of many low-cost embedded microprocessor-based systems.

To practically evaluate the feasibility of the approach, a software Fault Injection environment has been set up for a Motorola M68KIDP board. The preliminary results gathered on some simple benchmark programs have been reported to demonstrate the advantages of the approach.

Work is currently done to overcome the current limitations of the approach. In particular, we are working towards making it more efficient, by reducing the average time required to perform the analysis of each fault, and we are extending the described approach to a wider range of systems, e.g., those with real-time requirements, which can not be dealt with by the current version of the environment. The goal is to provide the user with a flexible environment, allowing him to select the most suitable Fault Injection technique, depending on the characteristics of the system and on the design requirements.

References

[1] M.C. Hsueh, T. Tsai, R.K. Iyer, *Fault Injection Techniques and Tools*, IEEE Computer, April 1997, pp. 75-82

[2] A. Benso, M. Rebaudengo, L. Impagliazzo, P. Marmo, *Fault List Collapsing for Fault Injection Experiments*, Proc. Annual Reliability and Maintainability Symposium, January 1998, pp. 383-388

[3] Motorola Inc., M68000 Family Integrated Development Platform (IDP), 1992, http://www.mot.com/SPS/HPESD/prod/68K_periph/68000IDP.html

[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, pp. 166-182

[5] J. Karlsson, P. Liden, P. Dahlgren, R. Johansson, U. Gunneflo, *Using Heavy-Ion Radiation to Validate Fault-Handling Mechanisms*, IEEE Micro, Vol. 14, No. 1, pp. 8-32, 1994

[6] E. Jenn, J. Arlat, M. Rimen, J. Ohlsson, J. Karlsson, *Fault Injection into VHDL Models: the MEFISTO Tool*, Proc. FTCS-24, 1994, pp. 66-75

[7] 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

[8] 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

[9] J. Carreira, H. Madeira, J. Silva, *Xception: Software Fault Injection and Monitoring in Processor Functional Units*, DCCA-5, Conference on Dependable Computing for Critical Applications, September 1995, pp. 135-149

[10] P.K. Lala, *Fault Tolerant and Fault Testable Hardware Design*, Prentice Hall Int., New York, 1985

[11] D. Powell, P. Verissimo, G. Bonn, F. Waeselynck, D. Seaton, *The Delta-4 Approach to Dependability in Open Distributed Computing Systems*, Proc. FTCS-18, 1988, pp. 246-251

[12] J. G. Silva, J. Carreira, H. Madeira, D. Costa, F. Moreira, *Experimental Assessment of Parallel Systems*, Proc. FTCS-26, 1996, pp. 415-424