A Fault Injection Environment for Microprocessor-based Board

A Fault Injection Environment for Microprocessor-based Boards

A. Benso, P. Prinetto, M. Rebaudengo, M. Sonza Reorda

Politecnico di Torino
Dipartimento di Automatica e Informatica
Torino, Italy

Abstract*

Evaluating the faulty behavior of low-cost microprocessor-based boards is an increasingly important issue, due to their usage in many safety critical systems. To address this issue, the paper describes a Software-implemented Fault Injection system based on the trace exception mode available in most microprocessors. The architecture of the complete Fault Injection environment is proposed, integrating modules for generating a fault list, for performing their injection and for gathering the results, respectively. Data gathered from some sample benchmark applications are presented. The main advantages of the approach are low cost, good portability, and high efficiency.

1. Introduction

Our society is facing 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. A major problem in the development of safety-critical systems is the accurate determination of the dependability properties of the system. Unlike performance, fault-tolerance and reliability can not 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 MTBF (*Mean Time Between Failure*) in a safety-critical 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 [ClPr95] emerged as a viable solution, and has been deeply investigated by both academia and industry. Several Fault Injection techniques have been proposed and practically experimented; they can basically be grouped into *simulation-based* techniques [JARO94] [DJPr96], *software-implemented* techniques [KKAb95] [CMSi95] [HSRo95], and *hardware-based* techniques [AAAC90] [KLDJ94]. As pointed out in [IyTa96], simulated fault injection is more suited for the early design phase, while physical fault injection (hardware- and software-implemented fault injection approaches) is more suited for the prototype and production phases of a system. The software-implemented approach can be effective when simple boards have to be analyzed, and hardware fault injectors, although generally less intrusive, are often too cumbersome and expensive.

This paper presents a software-implemented fault injection system, which is particularly suited for microprocessor-based boards. The main characteristics of the approach are the robustness, the reduced intrusiveness into the target system, the low cost (it does not require any special 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 kernel of the system is a *Fault Injection Manager*, which is based on the *trace exception mode* available in most microprocessors. A Trace procedure is automatically activated after the execution of every instruction, thus allowing the injection of the fault at the proper time, and the triggering of possible time-out conditions.

The overall system runs on two different units, connected by a serial port interface: a host computer and the actual target board. The communication

*Contact address: Matteo SONZA REORDA, Dipartimento di Automatica e Informatica, Politecnico di Torino, Corso Duca degli Abruzzi 24, I-10129 Torino (Italy), e-mail sonza@polito.it

0-7803-5092-8/98 $10.00

interface, the downloading of the code into the target board, and the analysis of the system behavior, exploit the routines available through the built-in ROM Monitor of the target board.

The tool is able to inject faults in the memory image of the process (data and code) and in the user registers of the processor.

The adopted fault model is the transient single bit-flip fault. This model is frequently used in fault injection tools [KKAb95] [DJPr96] since it is highly representative of faults occurring in real systems [Lala85]. Nevertheless, the approach can be easily extended to other fault models. Moreover, since Fault Injection aims at reproducing the behavior of actual faults that can appear in any moment during the system operation, the *injection time* is an additional degree of freedom that has to be taken into account while generating the fault list. Each fault is thus characterized by the following information:

- *fault injection time*: each fault is injected at the assembly level, before the execution of an instruction. The fault injection time is thus expressed in terms of number of instructions executed since the beginning of the application execution;
- *fault location*: the address of the memory location or the register where the fault has to be injected;
- *fault mask*: the bit mask that selects the bit(s) that has (have) to be flipped.

Therefore, for the purpose of the experiments described in this paper, each fault corresponds to flipping a single bit in a microprocessor register or in the memory area containing either the code or the data at a given time instant (e.g. executed instruction) during the program execution.

Our technique is ideally suited to systems whose behavior, in presence of a given sequence of input stimuli, can be deterministically computed and easily reproduced. Moreover, 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 development.

The approach resorts to Error Detection Mechanisms (EDMs) present in microprocessor-based systems: Hardware EDMs (i.e., system exceptions, built-in in the processor chip) and Software EDMs (i.e., software checks possibly inserted in the target application).

A case study is presented in which a Motorola M68KIDP board [Moto92] based on a M68040 microprocessor is considered; a prototypical version of a tool implementing the proposed approach has been setup, and some sample application programs are considered.

The paper is organized as follows: Section 2 describes the Fault Injection environment, and Section 3 reports some experimental results; some conclusions are eventually drawn in Section 4.

2. The Fault Injection System

As illustrated in Fig. 1, the fault injection system can be divided in three sections:

- the *Fault List Manager* (FLM) generates the fault list to be injected into the target system;
- the *Fault Injection Manager* (FIM) injects the faults into the target system;
- the *Result Analyzer* collects the results and produces a report concerning the whole Fault Injection experiment.

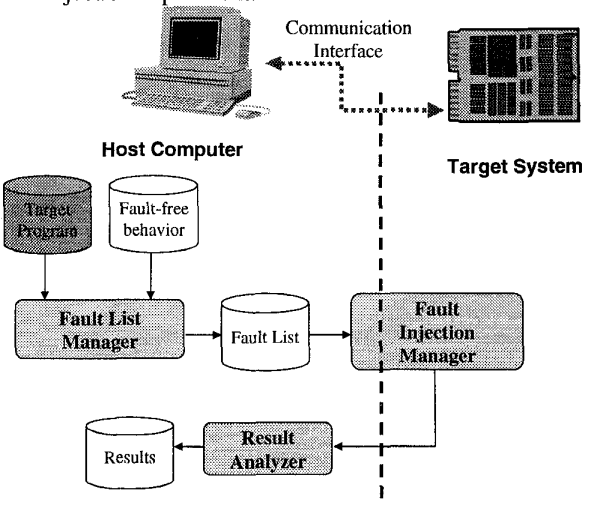

Fig. 1: The Fault Injection environment.

Since one of the main goals when setting up a Fault Injection environment is to minimize the intrusiveness into the target system, a host computer can be used to perform some of the tasks depicted in Fig. 1. The target system handles the serial communications exploiting the ROM monitor existing in most microprocessor system for debugging purposes. The host computer stores the relevant input and output information for the whole Fault Injection process (e.g., the Fault List, and the output statistics) and allows an easy interface towards the user, without introducing any overhead during the Fault Injection experiment. This solution presents several advantages:

- The intrusiveness into the target system is minimal. Only a small amount of additional code is present on the target system. Nevertheless, no code modification is required on the target application.

- The overall environment is more robust; in fact, if the target system crashes, the host computer can keep the control of the environment, reset the target system, and resume the execution of the experiment.
- The fault list and the result information are safe; they cannot be corrupted by a target system crash or by an error in the memory area storing them.

The main disadvantage of this approach is the slow-down factor caused by the communication through the serial interface. Nevertheless, this configuration can run the test of the target code in a working condition as close as possible to the real one.

2.1. The Fault List Manager

The *Fault List Manager* generates a random fault list according to some input constraints. To include in the proposed Fault Injection system more sophisticated fault list generation capabilities, we implemented a set of the collapsing rules presented in [BRIM98] and briefly outlined in the following. They aim at avoiding the injection of those faults whose behavior can be foreseen a priori. In particular, we can remove a fault from the Fault List when:

- the fault is guaranteed to trigger an Error Detection Mechanism;
- the fault is guaranteed not to have any effect on the target system behavior;
- the fault is equivalent to another fault already existing in the fault list.

From a practical point of view, the implementation of the fault collapsing rules requires the availability of some information collected during a preliminary fault-free run of the target program. In particular we generated an *instruction trace* that includes the *execution time*, the address, and the operative code of all the executed instructions. The execution time corresponds to the number of instructions executed from the beginning of the experiment. The number of executed instructions is also exploited to trigger possibly time-out conditions during the actual experiments.

Moreover, the system saves a copy of the data segment containing the results that it produced at the end of the fault-free run. This copy is then used to validate the application results at the end of each Fault Injection experiment.

2.2. The 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 and to continuously monitor the execution of the target application once for each fault in the fault list. When the fault injection time is reached, the fault injection is performed according to the fault type (e.g., single bit-flip) and location specified in the fault list. The pseudo-code of the FIM is reported in Fig. 2.

```
void Fault_Injection_Manager()
{
   for(every fault f_i in the fault list)
   {
     Environment_initialization(f_i);

     in parallel do
     {
       Execute_target_application();
       Inject_fault(f_i);
       Wait_for_completion();
     }

     Analyze_results();
   }
   return();
}
```

Fig. 2: Pseudo-code of the Fault Injection Manager.

Besides the target application code, in order to minimize the intrusiveness in the target system and to maintain the experiment as close as possible to the actual working conditions, only the `Inject_fault` and the `Wait_for_completion` modules are executed on the target board. All the other modules run on the host computer.

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

2.2.1. Environment initialization

This module is executed before the beginning of each new fault injection experiment in order to set up a fault-free environment where to execute the target code. The fault-free initial environment is necessary to avoid that the effects of a previous fault (e.g., corrupted bit in data and code memory sections) be still present in the environment where the new experiment is run.

The first task of the `environment_initialization` module is thus to download from the host computer to the target board a fault-free copy of the target application program in the memory area where the program is going to be executed.

To prevent its corruption during the experiments, the fault list is stored on the host computer. The second task of this module is thus to transfer on the target board the information concerning the fault to be injected and the time-out condition (e.g., the maximum number of instructions which can be executed before the time-out condition is activated.

2.2.2. Execute target application

This module starts the execution of the target code in trace mode. The Trace bit in the *Status Register* is enabled from the host computer using a ROM Monitor command. As depicted in Fig. 3, since the trace mode is enabled, after the execution of each assembly instruction a trace exception is triggered and the trace exception routine is executed. As explained in the following paragraphs, this routine is in charge of injecting the fault and triggering possible time-out conditions.

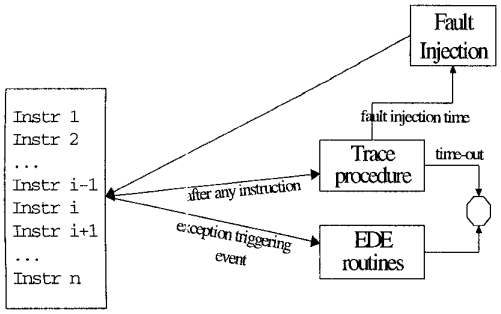

Fig. 3: Target code execution flow.

2.2.3. Inject fault

This module runs on the target board and is implemented by the trace exception routine. This routine is in charge of injecting the 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 current *injection time*, the procedure performs the injection, e.g., flips a bit in the target memory on in a target microprocessor register.

2.2.4. Wait for completion

After the injection of a fault the target code can behave mainly in three different ways:

- it can terminate without triggering any hardware EDMs, possibly with a failure in some output data result;
- it can trigger an hardware EDMs, and therefore force the execution of the relative exception routine;
- it can enter an endless loop or force the system in an unknown state, and therefore cause a time-out condition.

In the first case it is necessary to understand whether the fault caused a *Fail-Silent behavior* (i.e., the results are correct) or a *Fail Silent Violation Behavior* (i.e., the program terminated correctly but providing wrong results). Therefore, the `Analyze_results()` module should be 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. It is up to the application programmer to write this procedure, which is highly dependent on the application itself. In our environment, the results produced by the target code are just compared with the ones computed during the preliminary fault-free run, but more sophisticated check procedures can be used if needed.

To detect all faults triggering an exception during the system activity, we exploited the microprocessor built-in exception handling mechanism. We used the exception routines provided by the ROM Monitor present on the target board; these routines stop the execution of the target code and output an error message on the serial interface.

In the third case, again the Trace exception routine is exploited to monitor the instruction counter; if its value exceeds a user-defined limit the experiment is terminated and a message is output on the serial interface.

In all cases, the host computer captures the termination message from the target board, and initiates a new experiment.

2.3. The Result Analyzer

The *Result Analyzer* modules processes the system output behavior obtained through the Fault Injection experiments and produces a report concerning fault coverage information. The module runs on the host computer. 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.

3. Experimental results

To evaluate the effectiveness of our Fault Injection approach, a case study is described below.

The prototypical environment we considered is a commercial M68KIDP Motorola board [Moto92]. 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.

The Fault List Manager, the Fault Injection Manager and the Result Analyzer have been fully implemented in ISO-C and amount to about 2,000 source lines.

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: a program performing the multiplication of two matrices composed of 10x10 integer values.

For each target program, the 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).

Based on the fault list generated by the Fault List Manager, the Fault Injection Manager orchestrates the Fault Injection experiments, whose results are reported in Table 1.

Bubble Sort			
	Code	*Data*	*Regs*
Fault Category	%	%	%
Fail-Silent	58.10	66.11	70.81
Fail-Silent Violation	24.18	31.20	2.97
Detected by an EDM	15.75	2.20	17.09
Time-out	1.97	0.49	9.12
Parser			
	Code	*Data*	*Regs*
Fault Category	%	%	%
Fail-Silent	63.96	64.34	82.73
Fail-Silent Violation	13.42	10.34	2.99
Detected by an EDM	20.24	24.30	13.48
Time-out	2.38	1.02	0.80
Matrix			
	Code	*Data*	*Regs*
Fault Category	%	%	%
Fail-Silent	50.25	16.35	71.23
Fail-Silent Violation	25.15	81.55	3.18
Detected by an EDM	22.62	1.50	15.24
Time-out	1.98	0.60	10.35

Table 1: Faults injection report for the faults injected in the code, data and registers.

The results of Table 1 show 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 Fault Coverage figures concerning the whole fault list are reported in Tab. 2.

	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 2: Summary of Faults injection results.

Tab 3 reports the time required by for the whole fault injection experiment. The elapsed times required to perform the Fault Injection of 30,000 faults are reported, while the ones required by the Result Analyzer are always negligible and are not shown.

	Bubble Sort	*Parser*	*Matrix*
	[s]	[s]	[s]
Total Time	1,455	2,279	2,537

Table 3: Time required for the whole Fault Injection experiment.

To quantitatively evaluate the time required to perform a fault injection experiment using the proposed environment, we compared the total time reported in the last row of Table 3 with the one required to execute 30,000 time the same program with the same input data and without injecting any fault. The resulting ratio falls between 20 and 22 for the considered benchmarks.

The main cause of this slow-down factor is the time spent to exchange information through the serial interface, which is equal to about 80% of the total time needed for the whole experiment. Nevertheless, this configuration can run the test of the target code in a working condition very close the real one, with a minimal code overhead and intrusiveness, since from

the beginning of the target program execution to its termination, no communication is required on the serial line.

4. Conclusions

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

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

During each fault injection experiment, the target application program is executed in trace mode and the fault is 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.

References

[AAAC90] 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

[BRIM98] A. Benso, M. Rebaudengo, L. Impagliazzo, P. Marmo, *Fault-list collapsing for fault injection experiments*, Proc. Ann. Reliability & Maintainability Symp., 1998, pp. 383-388

[ClPr95] J. Clark, D. Pradhan, *Fault Injection: A method for Validating Computer-System Dependability*, IEEE Computer, June 1995, pp. 47-56

[CMSi95] 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

[DJPr96] 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

[HSRo95] S. Han, K.G. Shin, H.A. Rosenberg, *Doctor: An Integrated Software Fault-Injection Environment for Distributed Real-Time Systems*, Proc. IEEE Int. Computer Performance and Dependability Symposium, 1995, pp. 204-213

[JARO94] 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

[KKAb95] 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

[KLDJ94] 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

[IyTa96] R. K. Iyer and D. Tang, *Experimental Analysis of Computer System Dependability*, Chapter 5 of Fault-Tolerant Computer System Design, D. K. Pradhan (ed.), Prentice Hall, 1996

[Lala85] P.K. Lala, *Fault Tolerant and Fault Testable Hardware Design*, Prentice Hall Int., New York, 1985

[Moto92] Motorola Inc., M68000 Family Integrated Development Platform (IDP), 1992