EXFI: A Low-Cost Fault Injection System for Embedded Microprocessor-Based Boards

A. BENSO, P. PRINETTO, M. REBAUDENGO, and M. SONZA REORDA
Politecnico di Torino

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.

Categories and Subject Descriptors: B.8.1 [Performance and Reliability]: Reliability, Testing, and Fault-Tolerance; B.8.2 [Performance and Reliability]: Performance Analysis and Design Aids

General Terms: Design, Experimentation, Measurements

Additional Key Words and Phrases: Fault coverage, fault injection, microprocessor systems, software-implemented fault injection, trace exception mode

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. 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 [Hsueh et al. 1997] 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 (Exception-based Fault Injector), 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 exploits the collapsing rules defined in Benso et al. [1998], practically demonstrating how they can be implemented and how effective they really are.

A case study is presented, based on a prototypical version of EXFI implemented Motorola M68KIDP board [Motorola 1992].

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).

When compared with other Software-Implemented Fault Injection approaches, such as FERRARI [Kanawati et al. 1995], and Xception [Carreira et al. 1995], 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.

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

2. ASSUMPTIONS

The adopted fault model is the transient fault. This model is frequently used in Fault Injection tools [Kanawati et al. 1995; Delong et al. 1996], since it is very similar to the faults occurring in real systems [Lala 1985]. 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 mainly 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.

3. FAULT INJECTION SYSTEM

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.

3.1 Fault List Manager

The Fault List Manager (FLM) generates the list of faults that 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 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 Benso et al. [1998] 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 \textit{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.

To collapse the fault list, the Fault List Collapser exploits the information collected during a preliminary \textit{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.

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

—\textit{Identification of the fault injection time and injection of the fault}: 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.

—\textit{Fault Effects Observation}: The system behavior after fault injection must be observed, and differences with respect to the fault-free system behavior identified. This also requires the implementation of some time-out mechanism for the identification of faults forcing the system in endless loops.

—\textit{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.

The following paragraphs describe the different modules that compose the overall FIM module.
3.2.1 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.

3.2.2 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.

3.2.3 Trace Procedure. The Trace Procedure performs two main tasks:

—It monitors the system and 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.

3.2.4 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 faulty 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 execu-
tion 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.

3.2.5 Target Application Result Check Routine. A computer-based system is said to be Fail-Silent if it outputs only correct results, that is, if the system does not output incorrect results even if they are possibly generated internally as a consequence of a fault [Powell et al. 1988]. Many researchers [Silva et al. 1996] have shown that, in computer-based systems, a high percentage of faults cause a Fail-Silent Violation behavior, for example, 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 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.

3.3 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.

4. 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 [Motorola 1992]. This board hosts a M68040 microprocessor with a 25 Mhz 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 $10 \times 10$ 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 I. 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.

<table>
<thead>
<tr>
<th></th>
<th>Bubble Sort</th>
<th></th>
<th>Parser</th>
<th></th>
<th>Matrix</th>
</tr>
</thead>
<tbody>
<tr>
<td>Initial fault list size</td>
<td>10,000 10,000</td>
<td>10,000 10,000</td>
<td>10,000 10,000</td>
<td>10,000 10,000</td>
<td>10,000 10,000</td>
</tr>
<tr>
<td>Total removed faults</td>
<td>1,575 5,361</td>
<td>1,488 6,386</td>
<td>1,323 7,487</td>
<td>1,323 7,487</td>
<td></td>
</tr>
<tr>
<td>Detected by an EDM</td>
<td>792 127</td>
<td>671 154</td>
<td>788 139</td>
<td>788 139</td>
<td></td>
</tr>
<tr>
<td>No-effect</td>
<td>510 4,595</td>
<td>525 4,125</td>
<td>323 6,317</td>
<td>323 6,317</td>
<td></td>
</tr>
<tr>
<td>Fault Equivalence</td>
<td>273 639</td>
<td>292 2,107</td>
<td>212 1,031</td>
<td>212 1,031</td>
<td></td>
</tr>
</tbody>
</table>
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 Table II.

The results of Table II 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 Sort 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 III.

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 times 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.

5. 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. 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.
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, for example, 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


Table III. Summary of Faults injection results.

<table>
<thead>
<tr>
<th>Fault Category</th>
<th>Bubble Sort</th>
<th>Parser</th>
<th>Matrix</th>
</tr>
</thead>
<tbody>
<tr>
<td>Fail-Silent</td>
<td>60.62%</td>
<td>62.86%</td>
<td>32.09%</td>
</tr>
<tr>
<td>Fail-Silent Violation</td>
<td>26.35%</td>
<td>11.18%</td>
<td>52.19%</td>
</tr>
<tr>
<td>Detected by an EDM</td>
<td>11.98%</td>
<td>24.40%</td>
<td>14.54%</td>
</tr>
<tr>
<td>Time-out</td>
<td>1.06%</td>
<td>1.56%</td>
<td>1.19%</td>
</tr>
</tbody>
</table>