Fault Behavior Observation of a Microprocessor System through a VHDL Simulation-Based Fault Injection Experiment

*A.M. Amendola**, A. Benso*, F. Corno*, L. Impagliazzo**, P. Marmo**, P. Prinetto*, M. Rebaudengo*, M. Sonza Reorda**

*
Politecnico di Torino
Dipartimento di Automatica e Informatica
Corso Duca degli Abruzzi 24
I-10129 Torino, Italy

**
CRIS
Centro Ricerche Innovative per il Sud
Via Nuova delle Brecce 260
I-80147 Napoli, Italy

Abstract[1]

Evaluating and possibly improving the fault tolerance and error detecting mechanisms is becoming a key issue when designing safety-critical electronic systems. The proposed approach is based on simulation-based fault injection and allows the analysis of the system behavior when faults occur. The paper describes how a microprocessor board employed in an automated light-metro control system has been modeled in VHDL and a Fault Injection Environment has been set up using a commercial simulator. Preliminary results about the effectiveness of the hardware fault-detection mechanisms are also reported. Such results will address the activity of experimental evaluation in subsequent phases of the validation process.

1. Introduction

In recent years, there has been a rapid increase in the use of digital systems in critical real-time applications such as railway traffic control systems, aircraft flight, telecommunications, where computer resource failures can cost lives and/or money. This trend has led to concerns regarding the validation of the *dependability* properties of these systems. A dependable computer system is designed with the capability of: 1) detecting errors caused by hardware or software faults, 2) locating the cause of errors, 3) recovering from errors.

The validation process should provide [KKAb95]: 1) a measure of the ability of a system of detecting, locating, and recovering from errors, 2) confidence in the system before it is deployed, 3) a measure of the effectiveness of embedded fault tolerance mechanisms, and 4) a feedback during the development stage for improving the design and implementation of the system.

Fault injection has been recognized as an effective approach to evaluate the behavior and performance of complex systems under the effects of faults and to obtain parameters such as fault coverage and fault latencies. Fault injection is a powerful method to evaluate the dependability and the efficiency of fault-handling and fault-detection techniques implemented in a dependable computer system. More specifically, it can be used to evaluate error detection coverage and latency of employed error detection techniques while executing realistic programs.

A classification of the fault injection methods is based on the division into hardware-based (physical) [GKTo89] [AAAC90] and software-based fault injection [SVSY90] [KKAb95]. Furthermore, the software-based techniques can be separated in software-implemented fault injection, where data is altered and/or timing of an application is influenced by software while running on real hardware, and simulation-based fault injection, where the whole system behavior is modeled using simulation. Simulation-based fault injection can provide good control over the time and the location of the injected faults and good observability over the internal system state and behavior. Moreover, the simulation-based approach normally allows to avoid any undesired change (*intrusion*) in the examined system, due to the

[1] Contact address: Paolo Prinetto, Dipartimento di Automatica e Informatica, Politecnico di Torino, Corso Duca degli Abruzzi 24, I-10129 Torino (Italy), e-mail `Paolo.Prinetto@polito.it`

presence of the fault injection mechanisms, which is seldom possible with hardware- or software-based techniques. Two techniques have been proposed to inject faults into a simulated model [JARO94]: 1) *code alteration*, i.e., modifying the base components of the architecture or adding dedicated components to inject faults on the interconnections between different modules of the system, and 2) using the *simulator*, i.e., using the commands available through the simulator to modify the value of signals and variables of the system. Previous examples of simulation-based fault injection systems are NEST [DSYB90], DEPEND [GoIy93], REACT [ClPr93], MEFISTO [JARO94], and ADEPT [GJPr95].

This paper describes a simulation-based fault injection experiment on a real-time communication microprocessor architecture (IP module) used in automated light-metro systems. The experiment uses a VHDL model of the system and aims at reproducing the effects of faults located inside the processor chip and in the external memory, as well as faults affecting the busses. These errors are injected into the system using the *code alteration* technique. Based on the results of the fault injection experiments, we are able to evaluate the effectiveness of the existing hardware fault detection mechanisms: further analysis will then focus on faults which are not detected, and will verify whether they trigger higher-level software and hardware fault-tolerance mechanisms. The main contribution of this paper is the description of an industrial experience in the use of a commercial VHDL simulation environment for assessing the hardware reliability of a microprocessor-based control system.

The organization of the paper is as follows: Section 2 briefly describes the considered microprocessor system, Section 3 outlines the model we adopted for Fault Injection experiments, and Section 4 reports some information about the approach we followed to model the system using VHDL; Section 5 reports some preliminary results, and Section 6 draws some conclusions.

2. System Description

The microprocessor system we focus on in this paper is an Industry Pack (IP) module [Gree94] devoted to handling the serial communications of a hosting Motorola MVME162 Controller Board [Moto94].

The MVME162 Controller board is used in the new design of control equipments of automated light-metro systems. Communications between the central controller and the field unit controllers (lights, switches, sensors, etc.) are performed exploiting the serial channels managed by three IP modules.

Each IP module (Fig. 1) is equipped with a 16 MHz Motorola 68302 micro-controller, a 256 KByte dual-port SRAM memory, and some interface circuitry based on a Xilinx device. This circuitry mainly allows the 68040 hosting board processor to access the dual-port memory and interact with the 68302 processor through some shared interrupt lines.

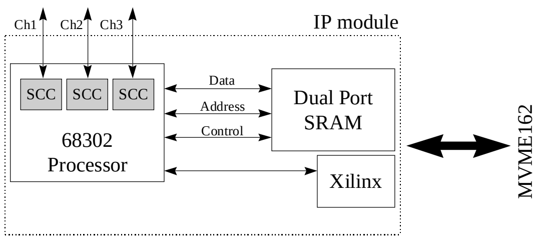

Fig. 1: Architecture of the IP module.

Each IP module handles three serial channels: several synchronous and asynchronous protocols are supported by the hardware. In our application, only the HDLC protocol is adopted; data are received from each channel and written in the SRAM, or read from the SRAM and sent to a channel. These operations are implemented in hardware for each channel through a specific part of the 68302 processor named Serial Communication Controller (SCC).

An initial hypothesis on the interaction with MVME162 is described in the following. The IP modules act as slaves of the 68040 processor of the hosting MVME162 board. The latter interacts with the IP module in two ways:

- at the bootstrap, by programming it: in this phase, the dual-port memory addresses are set up and the IP module software is loaded;
- during the normal behavior, by reading and writing data to and from the dual-port memory.

A simple program runs on each module, executing commands written by the master 68040 processor in the dual-port memory. Commands specify whether a send or receive operation must be performed, on which channel, and where data have to be written to or read from within the memory.

3. Building the Fault Injection Environment

The main goal of the Fault Injection experiments we performed is to evaluate which faults are not detected by the local fault detection mechanisms existing on the IP modules (e.g., exceptions generated by the 68302 processor). As a second goal, the experiments aim at classifying the possible faults according to the errors they produce and to their observability. This will allow us to verify whether these faults can be detected by higher level fault detection mechanisms, and to possibly modify the software in order to make it able to detect them.

3.1 The FARM Model

When designing a Fault Injection environment, it is necessary to take some decisions concerning the adopted Fault Models, the Activation patterns used to exercise the system, the set of Readouts gathered during the experiment, and the derived Measures. This set of parameters composes to the so called FARM set [AAAC90].

3.1.1 Fault Models

Due to their simplicity and wide acceptance, the following categories of faults have been taken into consideration:

- faults in the memory
- faults in the internal registers of the processor
- faults on busses.

In all the cases, the fault model consists of flipping the value of a single bit at a given time (*fault injection time*): for memory and register bits, the corresponding error perfectly matches the characteristics of errors produced by alpha particles.

3.1.2 Activation Patterns

During the experiment, the 68302 processor executes a program which is functionally similar to the one which will run on the real system, excluding higher-level software-implemented fault-tolerant mechanisms. The program performs different actions, according to the requests the 68040 processor makes by writing a suitable code in the IP module memory. The actions to be performed, as well as data received through the serial channels, and data to be sent through the same channels are randomly generated.

The main program running on the 68302 processor is composed of a simple polling cycle on the memory variable where the 68040 processor writes the code of the requested actions for the IP module. As soon as a request code is written, the module starts its execution, and then returns to the polling cycle.

3.1.3 Readout

We defined several *observation techniques* aimed mainly at defining which data have to be gathered during each experiment:

- *CPU status observation*: the status of the CPU (especially in terms of generated exceptions) is recorded; the occurrence of any exception stops the execution of the experiment
- *bus observation*: the values on the IP modules internal busses are continuously monitored
- *serial output observation*: the values sent on the serial channels are recorded
- *memory observation*: the contents of the memory elements (memory cells and CPU registers) are observed at the end of the Fault Injection experiment.

According to the adopted observation technique, relevant data are gathered during the Fault Injection experiment. Recorded data are then compared with the ones produced by the fault-free system.

3.1.4 Measures

On the basis of the gathered data we classified the injected faults according to the corresponding system behavior. A first class of faults does not produce any error in the system: on the other side, faults which produce errors can be grouped in the following classes:

- *hardware detected*: an *exception* is generated by the 68302, due either to a *bus error* (an address appeared on the bus, which does not correspond to any legal memory word) or to an *illegal instruction* (an instruction code which does not correspond to any legal instruction appeared on the data bus during the fetch operation)
- *latent*: the fault does not cause the occurrence of any exception, but at the end of the experiment is still *active*, i.e., a difference exists in the value of at least one memory element (memory cell or CPU register) with respect to the fault free system
- *bus active*: the fault causes at least one difference to appear on the bus.

When a fault is hardware detected, its detection time is recorded and its *latency* (e.g., the difference between the detection and fault injection time) computed.

4. Exploiting VHDL

4.1 Modeling the system

One of the most critical problems to be faced with when simulation-based Fault Injection is adopted is the need for suitable models of the system under consideration. As it often happens, in our case the models of the system components were not available, and we had to develop them in house. Note that pre-compiled models sold by third parties can not be of any help, as the Fault Injection procedure described below requires the modification of the source code. The models have been developed using VHDL language as defined by the 1987 IEEE Standard 1076.

In general, the abstraction level of the description must be detailed enough to accurately model the real system as far as the fault effects are considered; on the other hand, the description could not be too detailed, due to both the lack of structural information, and to cost in term of modeling effort and simulation time.

As a consequence, we developed a model with the following characteristics:

- the abstraction level is the one of Bus Operations: the main loop processes one bus operation per iteration; details about RT-level timing, internal temporary registers, and instruction microcoding are not represented
- delay information are not taken into any account.

The IP module description is composed of three main processes: the 68302 processor, the SRAM memory, and the Xilinx interface.

4.2 Injecting Faults

The experiments have been performed by simulating the execution of a simple but real program by the 68302 processor. Therefore, each experiment corresponds to reading an action code, sending or receiving data through a serial channel, and repeating for a while the polling cycle on the memory variable used by the 68040 processor to transmit the action code to the IP module.

To make the Fault Injection experiment possible, we made two kinds of modifications to the system description (Fig. 2). First, we added a *Bus Fault Injector* module, which is in charge of injecting faults at that level. Similarly, we inserted *Memory* and *CPU Fault Injectors* in the corresponding Memory and CPU modules to allow the injection of faults in the storing elements. The added parts are activated by a *Fault Injection Activator*, which can be programmed to inject one fault type (bus, memory, CPU), and randomly generates the specific location and time for the fault. Since no delay information is present in the model, the time for the injection is specified in terms of bus cycles.

During each experiment, all the readout data are gathered. A C program processes these data and computes the statistics on fault behavior.

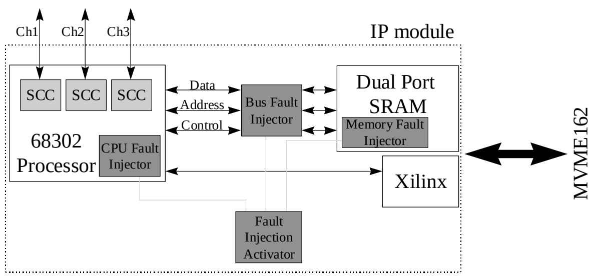

Fig. 2: Fault Injection Environment.

5. Experimental Results

All the experiments have been performed on a Sun SPARCstation 20/50 with a 64MByte memory. The Synopsis VHDL Simulator has been used.

Table 1 reports the data concerning the program described above: this program simply reads the action code sent by the 68040, and executes the corresponding send or receive operation. The code requires about 1,000 bus cycles to be executed. Writing and validating the system description required about two months for a skilled VHDL programmer; the resulting code amounts to about 4,000 statements.

The average time for injecting one fault and simulating the corresponding behavior of the system is about 20 msec.

In order to understand how results depend on the program characteristics, we developed a slightly different version of the same program, which additionally performs a very simple sorting operation on the data to be sent or received, and requires about 2,000 bus cycles to reach completion. Table 2 reports the results concerning this modified version.

The following considerations can be pointed out:

- the effects of changing the program significantly affect the distribution of faults among the different classes: in general, the higher the program activity, the larger is the number of faults producing an exception, or at least one difference on the bus values
- the most critical faults appear to be the ones injected in the memory, as a significant percentage of them remains latent without producing any visible error
- the results seem to contradict the assumption made in [KKAb93], that a significant percentage of the injected faults inside a processor are manifested as bus errors.

In Table 3 we report the average latency for the faults which generate an exception. The first program has been used for these experiments. There is evidence that for the faults injected in the bus the possible exception occurs very soon (e.g., due to an Illegal Instruction), while the occurrence is often much more delayed for faults injected in the memory.

Our current activity is aiming at gathering additional data, concerning in particular the analysis of the data transmitted on the serial channels.

6. Conclusions

The performance and versatility of commercial VHDL simulators allow their exploitation for the evaluation of the effectiveness of the fault-detection mechanisms of microprocessor systems. A state-of-the-art technique in this area is simulated Fault Injection: with this approach, it is possible to identify the faults able to escape to any existing detection mechanism, thus evaluating the system dependability, and possibly modifying the software to increase it.

The paper describes how we performed a set of Fault Injection experiments aimed at evaluating the dependability of a simple microprocessor-based system used in a railway control system.

The results show that a significant percentage of the injected faults trigger the intrinsic fault detection mechanisms of the microprocessor, although this ratio significantly changes according to the location where faults are injected (CPU, memory, bus).

The differences found in the results obtained through the two versions of the program push us to look for rules to be followed by software programmers in order to increase the system dependability.

Interestingly, our experiments showed that about one half of the injected faults never causes any difference to appear on the system bus, i.e., they either disappear very quickly after the injection, or remain latent in the memory elements. The latter class of faults is the most dangerous one from the point of view of system reliability, as they can cause further errors at latter time. Work is in progress to experimentaly assess the system higher-level fault-tolerance mechanisms. The results of the described experiments will be exploited during the subsequent phases in the design and validation of the system, as they will allow to focus on the malicious faults [SJPB95] only.

7. 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, Feb. 1990, pp. 166-182

[ClPr93] J.A. Clark, D.K. Pradhan, *REACT: A Synthesis and Evaluation Tool for Fault-Tolerant Multiprocessor Architectures*, Proc. Annual Reliability and Maintainability Symp., 1993, pp. 428-435

[DSYB90] A. Dupuy, J. Schwartz, Y. Yemini, D. Bacon, *NEST: A Network SImulation and Prototyping Testbed*, Comm. of the ACM, Vol. 33, No. 10, Oct. 1990, pp. 64-74

[GKTo89] U. Gunneflo, J. Karlsson, J. Torin, *Evaluation of Error Detection Schemes Using Fault injection by Heavy-ion Radiation*, Proc. FTCS-19, Chicago, USA, June 1989, pp. 340-347

[GoIy93] K. Goswami, R. Iyer, *Simulation of Software Behavior Under Hardware Faults*, Proc. FTCS-23, Toulouse, F, June 1993, pp. 218-227

[Gree94] GreenSpring Computers Inc., *User Manual IP-COMM302*, 1994

[JARO94] E. Jenn, J. Arlat, M. Rimen, J. Ohlsson, J. Karlsson, *Fault injection into VHDL Models: the MEFISTO Tool*, Proc. FTCS-24, June 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.

[KKAb93] G. Kanawati, N. Kanawati, J. Abraham, *EMACS: An Automatic Extractor of High-Level Error Models*, AIAA Computing in Aerospace Conference, San Diego, CA, October1993, pp. 1297-1306

[KKAJ94] S. Kumar, R.H. Klenke, J.H. Aylor, B.W. Johnson, R.D. Williams, R. Waxman, *ADEPT: A Unified System Level Modeling Design Environment*, Proc. RASSP'94, Arlington (VA), August 1994, pp. 114-123

[Moto94] Motorola Inc., *MVME162 Embedded Controller User's Manual - MVME162/D1*, 1994

[SJPB95] D.T. Smith, B.W. Johnson, J.A. Profeta III, D. G. Bozzolo, *A Fault-List Generation Algorithm for the Evaluation of System Coverage*, IEEE Annual Reliability and Maintainability Symposium, 1995, pp. 425-432

[SVSY90] Z. Segall, D.Vrsalovic, D. Siewiorek, D.Yaskin, J.Kownacki, J. Barton, R. Dancey, A. Robinson, T. Lin, *FIAT: Fault injection Based Automated Testing Environment*, Proc. FTCS-18, June 1988, Tokyo, pp. 102-107.

Fault Location	*Injected Faults*	*Error Activated*				
		Exceptions		*Latent*		*Bus Active*
	#	*Bus Error*	*Illegal Instr.*	*in the memory*	*in the CPU*	
Bus	1,000	250	47	120	97	607
CPU Reg.	1,000	109	18	227	537	364
Memory	1,000	8	6	885	64	94

Tab. 1: experimental results with the first program.

Fault Location	*Injected Faults*	*Error Activated*				
		Exceptions		*Latent*		*Bus Active*
	#	*Bus Error*	*Illegal Instr.*	*in the memory*	*in the CPU*	
Bus	1,000	276	36	28	28	552
CPU Reg.	1,000	100	33	133	545	335
Memory	1,000	5	5	870	75	120

Tab. 2: experimental results with the second program.

Fault Location	*Ave. Latency (#bus cycles)*
Bus	14
CPU Registers	68
Memory	178

Tab. 3: average latency for the first program.