Evaluating the Fault Tolerance Capabilities of Embedded Systems via BDM

M. Rebaudengo, M. Sonza Reorda

Politecnico di Torino
Dipartimento di Automatica e Informatica
Torino, Italy

Abstract*

Fault Injection is a viable solution for verifying the correct design and implementation of Fault Tolerance mechanisms at different levels (hardware and software). The paper discusses the use of the Background Diagnostic Mode (BDM), available on several Motorola microprocessors and microcontrollers, for implementing a Fault Injection environment. BDM is well suited to implement some of the most critical operations required by a Fault Injection environment, such as activating the injection procedure, injecting the fault in memory or registers, and observing the faulty system behavior. The characteristics of a BDM-based Fault Injection environment in terms of intrusiveness, flexibility, time efficiency, and system requirements are analyzed. The authors exploit a prototypical environment they implemented to validate this analysis. As a result, the approach appears to be well suited for implementing low-cost fault injection experiments on simple embedded microprocessor- and microcontroller-based boards. Some limitations are also outlined, mostly in terms of execution time slow-down.

1. Introduction

In recent years, there has been a rapid increase in the use of computer-based systems in areas such as railway traffic control, aircraft flight, telecommunications, and others, where failures can cost lives and/or money. This trend has led to concerns regarding the validation of the fault tolerance properties of these systems and the evaluation of their reliability.

On the other side, the continuous increase in the integration level of electronic systems is making more difficult than ever to guarantee an acceptable degree of reliability, due to the occurrence of unmodeled faults and soft errors that can dramatically affect the behavior of the system. As an example, the decrease in the magnitude of the electric charges used to carry and store information is seriously rising the probability that alpha particles and neutrons hitting the circuit could introduce errors in its behavior (often modeled as Single Upset Errors) [Nico98].

To face the above issues, mechanisms are required to increase the robustness of electronic devices and systems with respect to possible errors occurring during their normal function, and for these reasons on-line testing is now becoming a major area of research. No matter the level these mechanisms work at (hardware, system software, or application software), there is a need for techniques and methods to debug and verify their correct design and implementation.

Fault Injection [ClPr95] [IyTa96] [HTIy97] imposed itself as a viable solution to the above problems. Several Fault Injection techniques have been proposed and practically experimented; they can basically be grouped into *simulation-based* techniques (e.g., [JARO94] [DJPr96]), *software-implemented* techniques (e.g., [KKAb95] [CMSi95] [HSRo95] [BPRS98]), and *hardware-based* techniques (e.g., [KLDJ94]).

As pointed out in [IyTa96], simulated fault injection is more suited for the early design phases, or for small systems, while hardware- and software-implemented fault injection approaches are more suited when a prototype of the system is already available, or when the system itself is too large to be modeled and simulated at an acceptable cost.

The software-implemented approach is particularly effective when a fast solution to dependability evaluation problems is needed. In particular, when simple boards have to be analyzed, hardware fault injectors are generally too cumbersome and expensive.

The contribution of this paper is to show how to exploit some features available in some of the most recent microprocessors and microcontrollers to implement a *software-implemented fault injection* system suited to be used with embedded microprocessor-based boards. Although these features were originally introduced to easy code development and debugging, they are also well suited for implementing efficient and barely intrusive Fault Injection Systems. In particular, we discuss how the *Background Debugging Mode* (BDM) [Moto96] available in the last microprocessors and

* 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, http://www.cad.polito.it

microcontrollers produced by Motorola can be used to perform Fault Injection experiments. The architecture of a Fault Injection system is outlined, and the use of BDM for resetting the system, downloading the application target program, executing the fault injection, and triggering a possible time-out condition is described.

A case study is presented, which is based on a commercial tool interfacing a host computer with a Target board based on a MC68332 microcontroller. The behavior of such a board in presence of faults is evaluated using a prototypical implementation of the BDM-based fault injection approach. Preliminary experiments on some benchmark programs show that the approach has several nice properties: it does not depend on any Operating System facility, it is quite flexible (since it allows the injection of any fault affecting memory and/or registers), it is minimally intrusive in terms of code modification, and it has a very low cost, being easily portable from one system to another (provided that BDM is available). As a conclusion, the approach appears to be well suited to be used when the fault tolerance characteristics of low-cost embedded microprocessor-based systems have to be debugged and validated.

With respect to other software fault injector, such as *FERRARI* [KKAB95] and *Doctor* [HSRo95], our approach does not insert software traps or fault injection routines in the target software. Moreover, the target system is not supposed to provide either Operating System calls, such as the ones used in FERRARI to corrupt the process memory image, or any sophisticated exception routines, such as the ones exploited by the *Xception* tool [CMSi95], which are unique to the PowerPC processor. Finally, when compared with other approaches exploiting a software debug environment existing on the system, such as in *FITES* [BKAb98], our method has a much lower intrusiveness. The approach described in this paper shares several characteristics with that proposed in [BKAb98]: however, that system exploits a software debugger instead of BDM, therefore presenting requirements that are often not matched by low-cost embedded systems; moreover, a BDM-based Fault Injection environment guarantees a lower intrusiveness, and does not require re-compiling the application with the debugger library.

The paper is organized as follows: Section 2 briefly outlines the Background Debugging Mode. Section 3 describes its use when implementing a Fault Injection environment. Section 4 reports some experimental results gathered on our prototypical environment. Some conclusions are eventually drawn in Section 5.

2. BDM overview

The most recent Motorola's microprocessors and microcontroller devices feature a special mode of operation called Background Debugging Mode (BDM). When enabled, this mode allows an external host processor to control a target MicroController Unit (MCU) and access both memory and I/O devices via a simple serial interface.

For most microprocessor systems a software-implemented debugger is available for code development. With BDM the debugger is implemented in the CPU microcode. BDM can be very useful during initial debugging of control system hardware and software, and can also simplify production-line testing and configuration of an end product.

BDM uses a small amount of on-chip support logic, some additional microcode in the CPU module, and a dedicated serial port. By routing background mode interface signals to a simple board on the target system, the host debug computer takes complete control of the target system. BDM is an alternative microprocessor operating mode. When BDM is activated, normal instruction execution is suspended, and special microcode performs debugging functions under external control. The operations that can be executed in BDM are the classical operations available with a debugger tool: breakpoint activation, examining and changing registers and memory locations, single-stepping, resetting and restarting a program. None of the user resources of the chip or the target system, such as timers, on-chip memory, or I/O pins are required to support debugging mode. The microcontroller runs with no timing restrictions imposed by an external emulator.

The typical BDM configuration is reported in Fig. 1. The communication between the CPU on the target system and the host computer occurs via a dedicated serial interface (BDM port) which shares pins with other development features. When BDM is enabled, the pins belonging to the BDM port change functions to become a synchronous serial port. The CPU receives debug instructions via the BDM port, executes them, and returns results (if any) to the host system via the same BDM port.

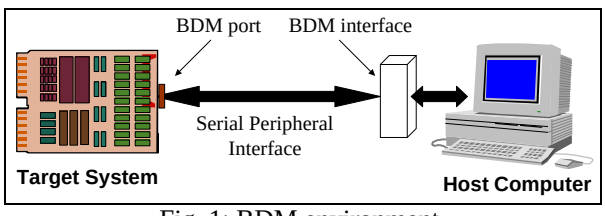

Fig. 1: BDM environment.

3. Fault Injection environment architecture

Major characteristics of a Fault Injection environment must be the ability to guarantee minimum intrusiveness into the original target program and to reach the maximum speed to allow efficient Fault Injection experiments. The architecture of the Fault Injection environment we refer to (shown in Fig. 2) is oriented to fulfill these requirements. This architecture is rather independent on the solution adopted to implement the Fault Injection mechanisms, and has been already exploited for other environments (e.g., the one described in [BPRS98]). Only the target application code is located on the target system, while the whole Fault Injection Manager runs on the Host Computer, communicating with the BDM port via the serial line. In this way, minimum code intrusiveness is guaranteed, and the Fault Injection Manager is largely independent on the target system which is currently considered.

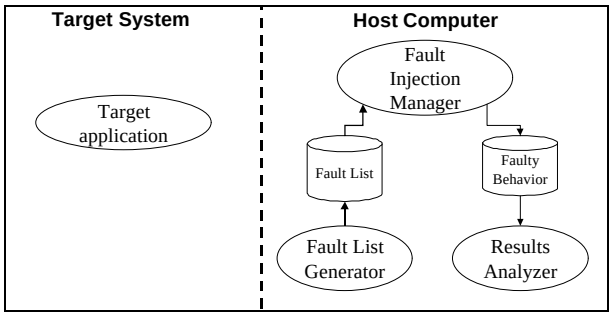

Fig. 2: Fault Injection system architecture.

3.1 Fault Injection Manager

In the following paragraphs the main general characteristics of the Fault Injection Manager (FIM) are presented. The pseudo-code describing its behavior is shown in Fig. 3. The main loop repeatedly processes one fault, taken from the fault list: for every fault, the FIM schedules the fault injection time and loads the environment on the target system (`target_system_initialization`); then, it spawns the target application program. While the application is running, the `inject_and_observe` process is active on the host computer. This process is in charge of monitoring the system, injecting the fault, and observing the system behavior. All the above listed operations are performed resorting to the corresponding BDM commands.

3.1.1 Target System Initialization

The initialization phase can be subdivided in two distinct steps:

- *target system initialization*, i.e., preparing the whole data area for the program and downloading the target program. This phase is repeated for every fault, therefore guaranteeing that any effect of the previously injected faults does not affect the program behavior during the analysis of the current fault.
- *fault injection data setup*: a fault is taken from the fault list, and a breakpoint is set in the code, so that the target program execution is interrupted when the time for injecting the fault is reached.

```
void fault_injection_manager()
{
  /* Experiment Control Loop */
  for(every fault f_i in the fault list)
  {
   target_system_initialization(f_i);
   spawn(target_application);
   inject_and_observe(f_i);
  }
  return();
}
```

Fig. 3: Fault Injection system architecture.

3.1.2 Recovery from fault effects

The Fault Injection system architecture should be able to recover from the effects generated by the injection of any fault, and this requires that the Fault Injection architecture maintains the system control even in the likely event of a hardware exception being triggered. To match this requirement, we propose to modify the *Error Detection Exception* (EDE) procedures which are available in any microprocessor system. In particular, the original version of an EDE procedure normally outputs an error message reporting the type of EDE that has been triggered. In our system, these procedures have been slightly modified, so that they also return the control of the target system to the FIM on the host processor. To perform this task, the procedure modifies the return address stored in the exception stack frame so that the return from the exception instruction returns the execution control to a specific instruction (`return_from_exception_instruction`) instead of to the instruction that triggered the exception. The BDM puts a breakpoint to the `return_from_exception_instruction`. Every EDE writes a different message in a global variable (`exception_type`). When the program activates an EDE and reaches the `return_from_exception_instruction`, the

FIM reads `exception_type` and classifies the fault accordingly.

In this way, no matter the type of exception triggered by the faults, it is possible to recover the error and start the injection of a new fault.

3.1.3 Fault Effects Observation

After the fault has been injected in the system, its behavior has to be observed, and the differences with respect to the fault-free system behavior have to be identified. When temporal constraints are not the main concern, this can be done by observing the values of some specified variables when a given point in the target program execution is reached.

Fault effects observation also requires implementing some time-out mechanisms for the identification of faults forcing the system in endless loops.

The behavior of the faulty system is observed and classified according to four main categories:

- *Fail-Silent (FS)*: the fault has no effect on the system behavior
- *Detected by some Error Detection Mechanism (D)*: the faulty system behavior triggered the activation of either a software or hardware EDM.
- *Fail-Silent Violation (FSV)*: the faulty system behavior does not trigger any EDM, but the output results are different from the fault-free ones
- *Time-out Violations (TO)*: the system triggers a time-out check when the time spent by the faulty program considerably exceeds the time needed by the fault-free execution.

To distinguish between faults belonging to the first and third categories, a suitable sequence of BDM commands has been included in the `monitor_and_inject` process. Preliminarily, a breakpoint is set each time a variable or register must be observed: every time one of these breakpoints is reached, a BDM command is activated, which accesses the variable or register value and verifies whether it corresponds to the fault-free value or not.

3.1.4 Time-out check

A fault could generate an error, which puts the program in an uncontrollable state (e.g., an endless loop). To avoid this state the program execution is controlled by means of a *watchdog* feature managed by BDM and programmed by the host processor. A BDM command sets the watchdog period to a time exceeding the one needed by the fault-free execution. If the program is still running when the watchdog period limit is reached, it is stopped, the fault is classified as “time-out”, and the experiment continues by injecting the next fault in the list.

4. Experimental Results

A prototypical version of the described Fault Injection environment has been implemented on the commercial Evaluation Board LA-7902 produced by Lauterbach GmbH. This board hosts a MC68332 microcontroller with a 16Mhz frequency clock, 128 kbytes of RAM memory and a V.24 interface. The BDM interface is managed by the TRACE32-ICD commercial tool produced by Lauterbach GmbH. The host computer is an 80486 PC with a 33Mhz frequency clock, 16 Mbytes of RAM memory, running Microsoft Windows95 Operating System.

The whole Fault Injection system is composed of about 500 lines of BDM program written in the PRACTICE language and running on the host computer. Apart from the module implementing the `observe_fault_behavior` function, the system can be easily adapted to deal with any target application program.

4.1 Fault Model

Although in the experiments we are describing we adopted a specific fault model, BDM-based Fault Injection can deal with other kinds of faults: in fact, any fault model that can be injected using assembly level instructions can be supported by our environment.

For the purpose of our experiments the transient fault model has been adopted, which is frequently used in fault injection tools [KKAb95] [DJPr96]. The fault type is the single bit-flip: faults are injected at the assembly level, i.e., the system injects every fault between one instruction and another. In this way, each fault can be deterministically identified and its effects easily reproduced [Stei98].

Each fault is identified by the following parameters:

- *Fault location*: the memory address (or user register) and the bit affected by the fault
- *Injection time*: the number of instructions executed from the beginning of the application execution.

To allow BDM to perform fault injection, the injection time is converted in the following format:

- *Instruction address*: the address of the instruction to be interrupted for fault injection
- *Instruction repetition*: the number of times *n* the considered instruction has to be executed before injecting the fault.

During the `target_system_initialization()` process a fault is taken from the fault list, and a breakpoint is

inserted in the code at the instruction corresponding to the fault. Thanks to this breakpoint, the `inject_and_observe()` process is activated at every execution of the instruction where the fault has to be injected. At its n-th activation, the process injects the fault by means of a BDM command that modifies the memory location or user register determined by the content of the fault location field. Other fault types can be easily implemented by modifying this single BDM command.

4.2 Fault List Generation

Fault List generation is performed before the whole Fault Injection experiment starts by executing the following steps:

- a fault-free run is executed to determine the time required for the execution of the whole program and to trace the list of executed instructions
- the original Fault List is randomly generated in terms of fault location and injection time
- the Fault List is converted to the format required by BDM, as described above.

Fault collapsing can possibly be exploited to reduce the size of the Fault List, e.g., by exploiting the rules introduced in [BRIM98], which identify and eliminate from the list several classes of faults, such as those that are known to trigger an Error Detection Mechanism, those that do not cause any change in the system behavior, and those that are equivalent to other faults.

We evaluated the fault coverage of the 68332-based board when executing the above programs by injecting three sets of faults, each composed of 1,000 randomly generated faults: one set includes faults located in the memory containing the data, another in the memory area containing the code, the third in the microcontroller registers.

4.3 Error Detection Mechanisms

Two classes of Error Detection Mechanisms (EDMs) are available in the applications we considered:

- Hardware EDMs, i.e., system exceptions, built-in in the processor chip (e.g., illegal instruction, divide by zero, address fault, access fault, bus error, privilege violation,etc.);
- Software EDMs, i.e., software checks, possibly inserted by the programmer in the target application.

4.4 Benchmark applications

To practically evaluate the characteristics of our system we adopted three benchmark applications:

- A simple program implementing the Bubble Sort algorithm, amounting to about 30 C language statements; we run it on an integer vector of 10 randomly generated elements.
- A program implementing a Parser for arithmetic expressions, and amounting to about 200 C statements; it includes some software checks on the correctness of the computed data.
- The Dhrystone Benchmark 2.1 [Dhry], originally developed for performance evaluation, amounting to more than 1,000 C code lines. Input data were randomly generated.

4.5 Input Stimuli

The input stimuli we used for our experiments have been randomly generated for the third benchmark, while correspond to functional test benches for the Bubble Sort and Parser programs.

4.6 Time requirements

Table 1 reports the time for performing a single execution of the fault-free program (*a*) and the time (*b*) required to perform the fault injection of the 3,000 faults defined above for each of the three programs. The experiments show that the analysis of each fault (*c*) requires from 70 to 96 times the time required to perform a single execution of the same program in normal mode operation.

	a	*b*	*c*
	Fault-free Execution [ms]	3,000 faults injection [s]	Ratio c=b/(a*3,000)
Bubble Sort	2.6	753	96.53
Parser	9.4	1,995	70.75
Dhrystone	20.6	4,950	80.09

Table 1: time requirements.

Please note that the time for each fault includes the one for recovery from previous faults effects, the execution of the program and the injection of the fault, and the observation of the output values. In particular, the time required is dependent on the threshold adopted for activating the time-out condition, which has been setup to twice the length of the fault-free run in our experiments. Moreover, the required time is proportional to the number of activated breakpoints, because each activation involves the exchange of some information between the target microcontroller and the host PC through the slow serial interface. Finally, a significant fraction of the time overhead (especially for short application programs) is due to the time for setting up the environment (including downloading the code) in the target system at every fault.

We are currently working to devise and implement a more efficient version of our system, allowing to strongly reduce the ratio *c*.

4.7 Fault Coverage Data

Table 2 reports the results collected for the three programs according to the classification introduced in Section 3.1.3. Differences in the results are due to the very different characteristics of the three programs: for example, the bubble sort program is very data intensive, while the Dhrystone benchmark, aiming at performance evaluation, does not necessarily exploit any computed value. However, please note that the goal of this paper is not to provide any evaluation about the fault coverage of the considered system, but simply to demonstrate the practical feasibility of a BDM-based fault injection approach.

Program	*Fault Location*	*Fault Effects*				*Total*
		FS	*D*	*FSV*	*TO*	
Bubble sort	data	257	2	741	0	1,000
	code	703	102	118	77	1,000
	registers	846	86	50	18	1,000
Parser	data	810	187	2	1	1,000
	code	406	222	359	33	1,000
	registers	836	77	83	4	1,000
Dhrystone	data	991	1	2	6	1,000
	code	664	71	233	32	1,000
	registers	807	64	111	18	1,000

Table 2: result summary

5. Conclusions

Effective Fault Injection techniques are required to validate the (hardware and software) Fault Tolerance mechanisms (including those for on-line testing) existing in most electronic devices and systems. This paper describes how the Background Diagnostic Mode (BDM) available in most Motorola microprocessors and microcontrollers can be exploited to implement a Fault Injection environment. A Fault Injection environment exploiting BDM is described: most of the code and data required to implement and manage Fault Injection are located on the host computer, therefore reducing intrusiveness on the target system and improving robustness. Moreover, being BDM a standard interface available on many different microprocessors and microcontrollers, a high level of portability can be easily achieved. The proposed solution is particularly suited for low-cost microprocessor-based embedded systems, where the limited support provided by the Operating System often prevents the adoption of other software-implemented Fault Injection techniques. The main limitation to the use of BDM for supporting Fault Injection is the slow-down it causes on the system behavior: the microprocessor speed decreases when BDM mode is entered, and the serial communications between the host computer and the target one introduce significant delays with respect to normal operations. As a result, the approach can not be easily applied to the validation of the time-related characteristics of real-time systems.

6. References

[BKAb98] R.P. Bulusu, N. Krishnamurthy, J.A. Abraham, *A Fault Injection Methodology for Embedded Systems*, IEEE International Computer Performance and Dependability Symposium, 1998, pag. 274

[BPRS98] A. Benso, P. Prinetto, M. Rebaudengo, M. Sonza Reorda, "*A Fault Injection Environment for Microprocessor-based Boards*," IEEE International Test Conf., 1998, pp. 768-773

[BRIM98] A. Benso, M. Rebaudengo, L. Impagliazzo, P. Marmo, "*Fault List Collapsing for Fault Injection Experiments*", Annual Reliability and Maintainability Symposium, 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, Urbana-Champaign, USA, 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

[Dhry] The code of the benchmark program can be downloaded from the URL http://www.amd-embedded.com/dhry2.htm

[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

[HTIy97] M.C. Hsueh, T. Tsai, R.K. Iyer, *Fault Injection Techniques and Tools*, IEEE Computer, April 1997, pp. 75-82

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

[Moto96] Motorola Inc., *A Background Debugging Mode Driver Package for Modular Microcontrollers*, by S. Howard, Motorola Semiconductor Application Note AN1230/D, 1996

[Nico98] M. Nicolaidis, "*Design for Soft-Errors Robustness to Rescue Deep Submicron Scaling*," Panel in IEEE International Test Conf., 1998, pag. 1140

[Stei98] A. Steininger, *How Reproducible should Fault Injection Experiments be?*, IEEE Fault Tolerant Computing Symposium, 1998, Digest of FastAbstracts, pp. 80-81