Journal of Universal Computer Science, vol. 5, no. 10 (1999), 693-711
submitted: 13/4/99, accepted: 22/10/99, appeared: 28/10/99

Fault Injection for Embedded Microprocessor-based Systems

Alfredo Benso
Politecnico di Torino
Dip. Automatica e Informatica
I-10129 Torino, Italy
E-mail: benso@polito.it

Maurizio Rebaudengo
Politecnico di Torino
Dip. Automatica e Informatica
I-10129 Torino, Italy
E-mail: reba@polito.it

Matteo Sonza Reorda
Politecnico di Torino
Dip. Automatica e Informatica
I-10129 Torino, Italy
E-mail: sonza@polito.it

Abstract: Microprocessor-based embedded systems are increasingly used to control safety-critical systems (e.g., air and railway traffic control, nuclear plant control, aircraft and car control). In this case, fault tolerance mechanisms are introduced at the hardware and software level. Debugging and verifying the correct design and implementation of these mechanisms ask for effective environments, and Fault Injection represents a viable solution for their implementation. In this paper we present a Fault Injection environment, named FlexFI, suitable to assess the correctness of the design and implementation of the hardware and software mechanisms existing in embedded microprocessor-based systems, and to compute the fault coverage they provide. The paper describes and analyzes different solutions for implementing the most critical modules, which differ in terms of cost, speed, and intrusiveness in the original system behavior.
Key Words: Fault Injection, Dependability Evaluation, Embedded microprocessor-based systems

1 Introduction

Our society is facing an increasing dependence on computing systems, especially in areas where a failure can be critical for the safety of human beings (e.g., air and railway traffic control, nuclear plant control, aircraft and car control). Safety-critical systems often incorporate levels of redundancy to guarantee the correct execution of the service in order to tolerate or simply signal the presence of possible faults that can cause system failures. The design and production flow of these systems must be different from the traditional ones, guaranteeing that the fault tolerant characteristics are correctly designed and implemented. For these reasons, new CAD tools and environments are required, which support the designer and production engineers in the task of realizing really reliable systems.

Fault tolerance and reliability measures can not be evaluated using benchmark programs and standard test methodologies, but only by observing the system behavior when a fault appears inside it. Since the 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.

Fault Injection (FI), the deliberate insertion of faults into an operational system to observe its response, has been recognized in [ClPr95] as a powerful technique that allows to validate some dependability characteristics of a system executing application programs.

Several Fault Injection techniques have been proposed and practically experimented; they can basically be grouped into *simulation-based* techniques [DJPr96], *software-implemented* techniques [CMSi95] [KKAb95] [Lovr95], *hardware-based* techniques [AAAC90], and *hybrid* techniques, where hardware and software approaches are applied together to optimize the performance [YoIG93].

None of the mentioned approaches seems to be a global solution in terms of costs, speed, and accuracy, since they are generally targeted to a particular platform or they consider only particular types of applications. Moreover, none of them is specifically targeted to embedded microprocessor-based systems, and they often exploit features (such as Operating System support) seldom available in these systems. Finally, the introduction of computer-based devices into safety-critical mass products (e.g., in the automotive area) requires new approaches to fault-tolerance evaluation, characterized by a lower cost and an easier applicability.

The goal of this paper is to analyze the main constraints that a FI environment must satisfy when dealing with microprocessor-based embedded systems, and to present a FI environment, called *FlexFI*, which can be easily customized to the specific needs of the embedded target systems. FlexFI includes different modules for Fault List Generation and Collapsing, Fault Injection, and Result Analysis. Most of the Fault Injection code runs on a host computer, which orchestrates the FI experiments and is connected to the target system exploiting existing features, normally provided for debugging purposes. Three versions of FlexFI are presented, based on a pure software solution, on a hybrid hardware-software approach, and exploiting the BDM feature existing in many of the most recent Motorola microprocessors and microcontrollers. The three versions are briefly described (greater details can be found in [BPRS98], [BCRS98], and [RSEO99], respectively) and their characteristics compared: they provide a full range of choices for the fault tolerance evaluation of an embedded system.

[Section 2] summarizes some background which is then exploited in the rest of the paper. [Section 3] outlines the general assumptions and decisions underlying the organization of the FlexFI system, [Section 4] describes its architecture, and [Section 5] outlines the characteristics of its three different versions. [Section 6] summarizes the characteristics of the three versions, and [Section 7] draws some conclusions.

2 Background

2.1 The FARM Model

Fault injection allows to validate dependability measures of a target system constituted by a *hardware architecture* and a *workload software application*.

A good approach to characterize a fault injection environment is to consider the FARM classification proposed in [AAAC90]. The *FARM* attributes are the following:

- the set of *faults F* to be deliberately introduced into the system
- the set of *activation trajectories A* that specify the domain used to functionally exercise the system
- the set of *readout R* that corresponds to the behavior of the system
- the set of *measures M* that corresponds to the dependability measures obtained trough the fault injection.

The FARM model can be improved by also including the set of *workloads W*.

The measures *M* can be obtained experimentally from a sequence of fault injection case studies. An *injection campaign* is composed of elementary injections, called *experiments*. In a fault injection campaign the input domain corresponds to a set of faults *F* and a set of activations *A*, while the output domain corresponds to a set of readouts *R* and a set of measures *M*.

The single experiment is characterized by a fault *f* selected from F and an activation trajectory *a* selected from *A* in a workload w from *W*. The behavior of the system is observed and constitute the readout *r*. The experiment is thus characterized by the triple <*f, a, r*>. The set of measures *M* is obtained in an injection campaign elaborating the set of readouts *R* for the workloads in *W*.

2.2 Fault Injection requirements

The FARM model can be considered as an abstract model that describes the attributes involved in a fault injection campaign, but it does not consider the fault injection environment, (i.e., the technique adopted to perform the experiments). The same FARM set can be applied to different fault injection techniques. Before presenting our solutions, we focus on the parameters that should be considered when setting up a Fault Injection environment.

A fault injection system can be evaluated according to its intrusiveness, speed, and cost.

2.3 Intrusiveness

The intrusiveness is the difference between the behavior of the original target system and that of the same system when it is the object of a Fault Injection campaign. Intrusiveness can practically be caused by:

- the introduction of instructions or modules for supporting FI: as an effect, the sequence of executed modules and instructions is different with respect to that of the target system when the same activation trajectories are applied to its inputs.
- changes in the electrical and logical setups of the target system, which result in a slow-down of the execution speed of the system, or of some of its components; this means that during the FI campaign the system shows a different behavior from the temporal point of view; we will call this phenomenon *time intrusiveness*.
- differences in the memory image of the target system, which is often modified by introducing new code and data for supporting the FI campaign.

It is obvious that a good FI environment should minimize intrusiveness, thus guaranteeing that the computed results can really be extended to the original target system.

2.4 Speed

A FI campaign normally corresponds to the iteration of a high number of FI experiments, each focusing on a single fault and requiring the execution of the target application in the presence of the injected fault. Therefore, the time required by the whole campaign mainly depends on the number of considered faults, and on the time required by every single experiment. In turn, this depends on the time for setting up the experiment, and on the one for executing the application in the presence of the fault.

The speed of the FI campaign can thus be improved by proceeding along one or both of the avenues of attack described in the following sub-sections.

2.4.1 Speeding-up the Single FI Experiment

The speed of a fault injection experiment is computed considering the ratio between the time required by the normal execution (without fault injection) and the average elapsed time required by a single fault injection experiment. The increase in the elapsed time is due not only to the time intrusiveness, but also to all the operations required to initialize the experiment, to observe the readouts, and to update the measures.

2.4.2 Reducing the Fault List Size

Since in a given time, the number of possible experiments is limited, a crucial issue when devising a Fault Injection environment is the computation of the list of faults to be considered. One challenge is to reduce the large fault space associated with highly integrated systems, improving sampling techniques and models that equivalently represent the effects of low-level faults at higher abstraction levels.

The fault list should be representative enough of the whole set of possible faults that can affect the system, so that the validity of the obtained results is not limited to the faults in the list itself. Unfortunately, increasing the size of the Fault List is seldom a viable solution due to the time constraints limiting the maximum duration of the Fault Injection experiment. In general, the goal of the Fault List generation process is to select a representative sub-set of faults, whose injection can provide a maximum amount of information about the system behavior, while limiting the duration of the Fault Injection experiment to acceptable values.

2.5 Cost

A general requirement valid for all the possible target systems is that the cost of the fault injection environment must be as limited as possible, and negligible with respect to the cost of the system to be validated.

We can consider as a cost the following issues:

- the hardware equipment and the software involved in the fault injection environment
- the time required to set up the fault injection environment and to adapt it to the target system.

The first issue is strictly related to the fault injection technique chosen, whereas the second one implies to define a system as flexible as possible that can be easily modified when the target system is changed, and can be easily used by the engineers involved in the fault injection experiments.

3 Assumptions

In this Section we report the assumptions (in terms of the FARM model described in [Section 2]) and choices underlying the organization of the FlexFI system.

3.1 Set F

It is the set of faults to be injected in a fault injection campaign. First of all, the fault model has to be selected. This choice is traditionally made taking into account from one side the need for a fault model which is as close as possible to real faults, and from the other side the practical usability and manageability of the selected fault model. Based on these constraints, the fault model we selected is the transient single bit flip. This model is frequently used in fault injection tools [KKAb95] [DJPr96] since it is very similar to the faults occurring in real systems [Lala85].

In the present version, each fault is 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. This choice clearly limits the set of time instants when a fault can be injected, but makes easier obtaining full repeatability of each Fault Injection experiment [Stei98].

- *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, 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. A *golden run* experiment is performed in advance and is used as a reference for fault list generation and collapsing. The golden run can be obtained assuming a deterministic environment, whose behavior can be deterministically determined when the input stimuli are given.

Although the fault model adopted in the current version of FlexFI is the transient single bit flip in memory cells and microprocessor registers, FlexFI can support a large set of other fault models (e.g., bridging faults, multiple bit-flip faults, stuck-at faults), provided that they can be injected through a software procedure.

Faults are located in memory cells and CPU registers, but they do not mimic only faults in this parts of the system, since they can be equivalent to faults in other parts of the system such as in the busses, the arithmetic units and other functional units. For example, a fault that changes the operand of an add instruction is equivalent to a fault in the cell containing the operand as well as a fault in the arithmetic unit.

The size of the fault list is a crucial parameter for any kind of Fault Injection experiment, because it dramatically affects the feasibility and meaningfulness of the whole Fault Injection experiment. For this reason, our environment includes a module for *fault list collapsing*, which is based on the techniques presented in [BRIM98]. The rules used to reduce the size of the fault list do not affect the accuracy of the results gathered through the following Fault Injection experiments, but simply aim at avoiding the injection of those faults whose behavior can be foreseen a priori. The validity of the collapsing rules is bounded to the specific Fault Injection environment which is going to be used, and to the set of input stimuli the target system is going to receive.

A fault can be removed from the fault list when it can be classified in one of the following classes:

- it affects the operative code of an instruction and changes it into an *illegal operative* code; therefore, the fault is guaranteed to trigger an *Error Detection Mechanism* when the instruction is executed (possibly provided by the processor);
- it affects the code of an instruction after the very last time the instruction is executed, and it is thus guaranteed not to generate *any effect* on the program behavior;
- it affects a memory location containing the program data or a microprocessor register before a write access or after the very last access; it is thus guaranteed not to generate *any effect* on the program behavior;
- it corresponds to flipping the same bit of the code of an instruction than another fault, during the period between two executions of that instruction; the two faults thus belong to the same *equivalence class*, and can thus be collapsed to a single fault;

- it corresponds to flipping the same bit of a memory location containing the program data, or a microprocessor register during the same period between two consecutive accesses of that location than another fault; the two faults thus belong to the same *equivalence class*, and can thus be collapsed to a single fault.

Experimental results gathered with some benchmark programs show that the average reduction in the fault list size obtained applying the proposed collapsing techniques is about 40% [BRIM98], considering an initial fault list composed of a random distribution of faults in the data memory, code memory, and processor registers.

3.2 Set A

Two important issues relate to this point:

- how to determine an input trajectory to be applied to the target system during each FI experiment; several proposals have been made to solve this general problem. In this paper, we do not deal with this problem, but we limit our interest to the techniques for performing the FI campaign, once the trajectory is known.
- how to practically apply the trajectory to the system; this issue is particularly critical when considering embedded system, since they often own a high number of input signals of different types (digital and analog, high- and low-frequency, etc.). In general, the only viable solution requires setting up a suitable environment around the target system, able to excite it with the right stimuli. This environment is often the same that is used for performing the behavioral and final testing of the system.

3.3 Set R

This set of information is obtained observing the system behavior during each fault injection experiment, and identifying the differences with respect to the fault-free system behavior. Note that all the operations involved by the observation task should also be minimally intrusive. In the FlexFI environment all the operations on the target system are controlled by an external host computer that compares the final state of the system and the output values with the correct ones computed with the golden run.

By incidence, to evaluate the faulty behavior requires implementing some time-out mechanism for the identification of faults forcing the system in endless loops, or forcing it into deadlock conditions.

3.4 Set M

At the end of the FI campaign, a proper tool should build a report concerning the dependability measures and fault coverage computed on the whole Fault List. Fault coverage is defined with respect to some Error Detection Mechanism (EDM). Microprocessor systems usually provide some mechanisms to detect faults, such as:

- Hardware EDMs, i.e., system exceptions, built in the processor chip;
- Software EDMs, i.e., software checks, possibly inserted in the target application.

Faults are classified into one of the following categories:

- *No output errors*: the fault does not produce any failure;
- *Detected by an EDM*: the fault triggers a system Exception (such as *illegal instruction*, *divide by zero*, *address fault*, *access fault*, *bus error*, *privilege violation*) or a software EDM. Upon exception triggering, an *exception routine* is executed to possibly recover from the error or to halt the system.
- *Fail-Silent Violation* behavior: the target application terminates correctly but produces incorrect results;
- *Time-out*: the number of executed instructions exceeds a user-defined limit (e.g., because the target application entered into an endless loop).

4 The Fault Injection Environment

4.1 Overall Architecture

The architecture of the FlexFI fault injection environment is shown in [Fig. 1]. The system is logically composed of the following main modules:

- the *Fault List Manager* generates the fault list to be injected into the target system;
- the *Fault Injection Manager* injects the faults into the target system;
- the *Result Analyzer* analyzes the results and produces a report concerning the whole Fault Injection campaign.

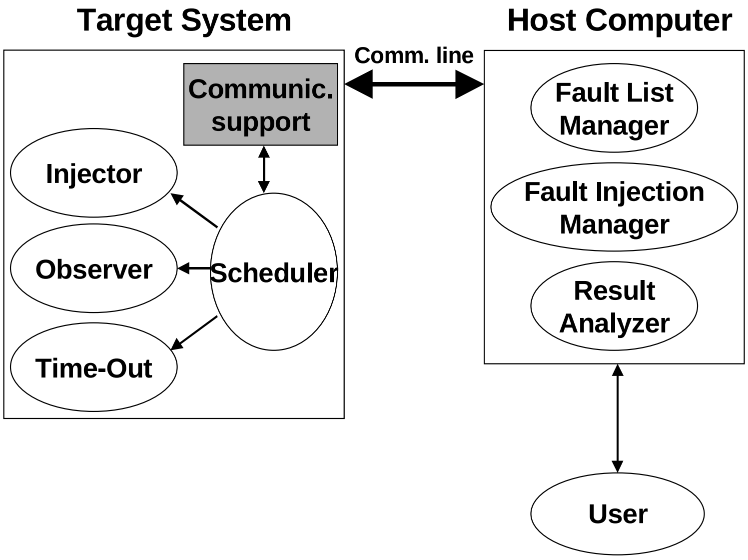

Figure 1: The architecture of the FlexFI system.

To minimize the intrusiveness into the target system, the FlexFI system uses a host computer. All the FI tasks which are not strictly required to run on the target system are located on the host computer, which also stores all the data structures (e.g., the Fault List and the output statistics) required by the FI campaign. The host computer communicates with the target system by exploiting the features provided by most systems for debugging purposes (e.g., the serial line handled by a ROM monitor which allows the debugging of most microprocessors).

4.2 Fault Injection Manager

The Fault Injection Manager (FIM) is the most crucial part in the whole Fault Injection System. In fact, it is up to the FIM to start the execution of the target application once for each fault of the list generated by the Fault List Manager, to inject the fault at the required time and location, and to observe the system behavior, recovering from any possible failure (e.g., from hardware generated exceptions). The pseudo-code of the FIM is reported in [Fig. 2].

```
void fault_injection_manager()
{
    campaign_initialization();

    /* Experiment Control Loop */
    for (every fault fi in the fault list)
    {
        experiment_initialization(fi);

        spawn(target_application);
        spawn(F_I_scheduler);

        wait for experiment completion;

        update_fault_record(fi);
    }

    return();
}
```

Figure 2: Fault Injection Manager pseudo-code.

During the target application execution, a *FI scheduler* monitors the advancement of the target program, triggering other FI modules in charge of injecting the fault (*Injector* module), observing variable values in order to classify the faulty behavior (*Observer* module), or stop the target application when a time-out condition is reached (*Time-out* module).

The pseudo-code of the FI scheduler module is reported in [Fig. 3]. Note that the Observer module refers to an ad hoc data structure, which contains the list of observation points; for each point, this data structure contains the name of the variable, the time when the variable should be observed, as well as the value the variable should have at that time. The list must be filled by the application programmer based on the knowledge of the behavior of the application itself.

```
void F_I_scheduler()
{
 instr_counter++;

 if (instr_counter==fault.time)
      trigger(injector());

 for (i=0; i<num_of_observation_points; i++)
    if (instr_counter==observation_time[i])
    trigger(observer(observed_variable[i], value[i]));

 if (instr_counter>max_time)
      trigger(time_out());
}
```

Figure 3: Pseudo-code of the Scheduler module.

In order to allow the FIM to maintain the control over the FI campaign, a mechanism has to be devised and implemented to handle the case, in which a hardware exception is activated, and the target application is consequently interrupted. The target system Exception handling procedures have to be suitably modified for this purpose, so that they first communicate to the FIM the type of triggered exception, and then return the control to it (instead of the interrupted instruction).

It is worth underlying the importance of the experiment initialization phase: the effects of the fault injected during an experiment should never affect the behavior of the target application when the following experiment is performed; for this reason, the FI system must restore the environment for the target application execution as a preliminary phase of each experiment. One safe (but slow) way to do so is to restore the full memory image of the application (code and data) and the values of all the relevant system variables. The main issue when implementing this restoring task is to limit its time duration as much as possible, in order to reduce the time requirement of the global FI campaign.

In the following, we will present different techniques for implementing these modules in an embedded system.

5 Implementation Issues

After having described the general architecture of the FlexFI system, we now propose three different solutions for implementing it, taking into account the constraints specified in [Section 3]. The three solutions differ in terms of intrusiveness, speed, and cost, thus offering a full range of possible choices for dealing with fault-tolerant embedded systems:

- *Software-based solution* [BPRS98]: it adopts a software-based technique which exploits the trace exception mode available in most microprocessors; it represents a low-cost approach, whose main limitations are the time intrusiveness and the relatively high slow-down factor.

- *Hybrid solution* [BCRS99]: it is based on a hybrid technique in which faults are injected via software by means of an interrupt procedure triggered by an extra hardware board; time intrusiveness is very limited in this case, at the cost of developing a custom hardware module devoted to FI.
- *BDM-based solution* [RSEO99]: it exploits the Background Diagnostic Mode [Moto96] existing in many of the most recent microprocessor and microcontroller kernels produced by Motorola. The advantages of hardware solutions are reached in this case by simply exploiting a feature provided for free by the processor. Nevertheless, being customized on a particular microprocessor feature, porting this approach on a different platform can be a very difficult task.

Each of the three solutions has been implemented in a prototypical version to practically evaluate its characteristics. Details are now provided on the resulting prototypical tools.

5.1 Software-based Solution

5.1.1 Description

This solution exploits the Trace Mode facility existing in most microprocessors for implementing the FI scheduler: thanks to the trace mechanism, a small procedure (corresponding to the FI scheduler) can be activated after the execution of any application assembly instruction with minimum intrusiveness in the system behavior (apart from a slow-down in the application performance). The proposed approach is similar to the ProFI tool [Lovr95], with the main difference that the fault injection experiment is completely executed by the microprocessor without any simulation.

The *FI scheduler* procedure is in charge of counting the number of executed instructions and verifying whether any FI module reached its activation point. When proper, the procedure activates one of the following modules, each corresponding to a software procedure stored on the target system:

- the *Injector* module, which is activated when the fault injection time is reached.
- the *Time-out* module, which is activated when a predefined threshold in terms of number of executed instructions is reached, and stops the target application, returning the control to the FIM located on the host.
- The *Observer* module, which is in charge of observing the value of target application variables, thus checking whether the application is behaving as in the fault-free fashion or not. When differences are observed, these are communicated to the FIM through the serial interface. The observer module is activated at proper times, depending on the target application characteristics.

5.1.2 Prototypical Version

We implemented a software-based version of FlexFI for a commercial M68KIDP Motorola board [Moto92b]. 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. To guarantee a deterministic behavior the internal caches have been disabled during the FI campaign.

The Fault Injection Manager is composed of the scheduler procedure, which amounts to about 50 Assembly code lines, of the modified Exception handling routine, which needs about 10 Assembly code lines more than the original one, and of the Initialization procedure, which is written partly in ISO-C and partly in Assembly language and globally amounts to about 200 source lines. Due to the high modularity of the FIM code, the task of adapting it to a new application program can easily be accomplished.

When run on some sample benchmark applications, this version of FlexFI showed a slow-down factor due to Fault Injection of about 25 times.

5.1.3 Comments

The software-based version of FlexFI is the most general one (the approach can be implemented on virtually any system) and does not require any special hardware, thus being very inexpensive.

On the other side, this approach has some drawbacks:

- there is some code intrusiveness, due to the need for storing the scheduler procedure, as well as the Injector, Observer, and Time-out procedures, in the target system memory
- there is also some data intrusiveness, since some small data structures, such as the one for storing the information about the current fault and the observation points, must also be stored in the target system memory
- forcing the target system to work in Trace mode causes a very high degradation in the execution speed of the application program; thus preventing this approach from being used with real-time embedded systems.

5.2 Hybrid Solution

5.2.1 Description

This solution overcomes the major drawbacks of the previous software-based solution, at the cost of introducing some custom hardware. An external board is exploited during the Fault Injection experiment to implement the scheduler.

The board is equivalent to a low-cost and specialized logic-analyzer. It is connected to the target system bus and is able to count the number of executed instructions by monitoring the values on the *processor status pins*. When one of the pre-defined points is reached, the board activates an interrupt protocol and triggers the proper FI module. Running concurrently with the target processor, the board avoids the overhead introduced by the software-based version. Therefore, the execution speed of the target system is not changed during the FI experiment, and the environment can be effectively exploited for evaluating real-time applications.

The board can work in two different modes:

- in *off-line mode*, it acts as a peripheral device, and can properly receive and react to read and write commands from the target system CPU
- in *on-line mode*, it continuously monitors the processor status pins, and counts the number of executed instructions from the last `start` command; as soon as the instruction counter matches the *injection time*, the board sends an interrupt to the processor. The interrupt handling routine is in charge of injecting the fault.

During the experiment initialization phase, the host computer must program the board by sending it the following commands:

- `set_injection`: it defines the fault injection time, i.e., the number of executed instructions before the fault injection
- `set_timeout`: it defines the time-out threshold, i.e., the maximum number of instructions that can be executed before stopping the experiment
- `set_observation_point`: it defines an observation point, corresponding to a 3-uple composed of an observation time (in terms of number of instructions), a variable address, and a variable value.
- `start`: the board begins to count the instructions executed by the processor
- `stop`: the board becomes idle and waits for other commands to start the next experiment.

At the end of every fault injection experiment (no matter its result), the control returns to the Fault Injector Manager, which classifies the fault according to the observed system behavior and updates the statistics stored on the host computer.

5.2.2 Prototypical Version

We implemented a prototypical version of the board, which is customized for a MC68040 microprocessor. The FI board has then been evaluated on the same commercial M68KIDP Motorola system used for the software-based version of FlexFI described in the previous subsection. In our implementation, the board is a memory-mapped device, thus allowing the CPU to program and control it through simple memory writes and read instructions. The board has been implemented using two Programmable Logic Devices, thus guaranteeing its re-programmability and flexibility. The board includes 2 Xilinx XC3130 FPGAs, a 256K x 56 bits memory, and some glue logic. Although in the current version the board has been customized for a MC68040 microprocessor, the same approach can be followed for other microprocessors, provided that they make status information available through the pins.

When run on sample benchmark applications, the hybrid version of FlexFI showed a slow-down factor which was almost exclusively due to the time required by the experiment initialization phase, thus strongly dependent on the size of the memory image of the target applications, and on the speed of the communication link between the target system and the host computer. The external board does not introduce any slow-down on the system clock of the target system, thus it is not time intrusive.

5.2.3 Comments

The intrusiveness of the fault injection process into the target system from the time point of view is practically removed so that this version of the FlexFI system can be adopted to deal with real-time embedded systems.

On the negative side, this solution requires the design and implementation of a hardware device which must match the characteristics of a specific target system.

5.3 BDM-based Solution

The most recent Motorola's microprocessors and microcontroller devices feature a special mode of operation called Background Debugging Mode (BDM) [Moto96]. When enabled, this mode allows an external host processor to control the internal microcontroller unit (MCU) and access both memory and I/O devices via a simple serial interface. BDM was originally introduced to easy code development and debugging, but is also well suited for supporting the implementation of efficient and barely intrusive Fault Injection Systems.

During the fault injection experiment, the application program is executed in debugging mode and BDM is in charge of resetting the system, downloading the application target program, executing the fault injection, and triggering a possible time-out condition. The method allows the injection of faults both in the memory image of the process (data and code) and in the internal registers of the processor.

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 experiment initialization, the FIM sets a breakpoint at the instruction corresponding to the considered fault. Thanks to the above mentioned breakpoint, the *scheduler* process is activated at every execution of the instruction where the fault has to be injected. At its n-th activation, the scheduler activates the *injector*. Note that in this version the scheduler is a software module running on the host computer, while the injector corresponds to a single BDM command that modifies the memory location or user register determined by the content of the fault location field.

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. A suitable sequence of BDM commands has been included in the *observer* module. Preliminarily, a breakpoint is set each time a variable or register must be observed: every time one of these breakpoints is reached, the scheduler is activated, that in turns issues a BDM command, which accesses the variable or register value and verifies whether it corresponds to the fault-free value or not.

Also the *Time-out* module is 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.

5.3.1 Practical Experience

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, 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 *observer* module, the system can be easily adapted to deal with any target application program.

When running it on the usual benchmark applications, we observed two kinds of slow-down phenomena. The first one is due to the time required by the experiment initialization phase, and is dependent on the size of the memory image of the target application, which is downloaded on the target system before every FI experiment starts. The second one is due to the fact that forcing the microprocessor into the BDM mode causes its clock to slow-down by a factor of about 2 times.

5.3.2 Comments

The approach is minimally intrusive in terms of code modification: the only required modification on the application software is the one concerning Exception procedures. It is also easily portable from one system to another, provided that BDM availability is given. Similar (and even more general) debugging features are supported by other microprocessor families, and can effectively be exploited for Fault Injection purposes [CMSi95].

A certain slow-down in the execution time of the target application can be observed, mainly due to the slow-down on the system clock forced by the activation of the BDM mode. Due to this fact, this approach can hardly be exploited with real-time applications.

6 Summary

In this Section we provide the reader with a comparative overview of the different versions of the FlexFI system, gathering the results of the practical experiences we made with our prototypical system.

First of all, [Tab. 1] summarizes the ways the critical modules of the FIM are implemented in the three versions. FW stands for firmware, being the BDM commands implemented in the microcode of the processor. Experiment initialization mainly aims at rebuilding the proper environment for the fault to be injected, by downloading from the host computer the target application memory image in the system and setting up the system variables.

	FI scheduler	*Injector Observer Time-out*	*Experiment Initialization*
SW-based	SW	SW	ROM Monitor
Hybrid	HW	SW	ROM Monitor
BDM-based	FW	FW	FW

Table 1: implementation solutions for the main modules of the FIM.

[Tab. 2] summarizes the main characteristics of the three versions in terms of cost (for equipment and for development), intrusiveness, and speed.

	Cost		*Intrusiveness*	*Speed*
	for equipment	*for development*		
SW-based	Low	Medium	High	Low
Hybrid	Medium	High	Low	High
BDM-based	Low	Low	Medium	Medium

Table 2: comparing the characteristics of the three versions of the FlexFI system.

[Tab. 2] shows that the three versions of the FlexFI system, although easily interchangeable within the environment, have complementary characteristics, thus providing the designer with a high flexibility in choosing solution which is best suited to his needs.

7 Conclusions

When evaluating the fault tolerance mechanisms of embedded microprocessor-based systems used in safety-critical applications, the designer needs a suitable environment allowing to effectively perform FI experiments.

The paper describes the architecture of the FlexFI system, which is particularly suited for embedded microprocessor-based systems, and whose main characteristic is that it is customizable to the specific needs of the considered application. In particular, we presented three versions of FlexFI, which allow the designer to chose the best solution in terms of cost, intrusiveness, and speed of the FI experiments. Prototypical implementations of the three versions have been built to verify their feasibility and effectiveness, and a comparison between their characteristics has been reported.

Further work is currently being done to improve FlexFI from the user friendliness point of view, and to evaluate it on other applications.

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, pp. 166-182

[ABCI96] A. M. Amendola, A. Benso, F. Corno, L. Impagliazzo, P. Marmo, P. Prinetto, M. Rebaudengo, M. Sonza Reorda, *Fault Behavior Observation of a Microprocessor System through a VHDL Simulation-Based Fault Injection Experiment*, EURO-VHDL'96, September 1996, Geneva (CH), pp. 536-541

[BCRS99] A. Benso, P.L. Civera, M. Rebaudengo, M. Sonza Reorda, *A low-cost programmable board for speeding-up Fault Injection in microprocessor-based systems*, RAMS'99: Annual Reliability and Maintainability Symposium, Washington, DC (USA), January 1999, pp. 171-177

[BPRS98] A. Benso, P. Prinetto, M. Rebaudengo, M. Sonza Reorda, *A Fault Injection Environment for Microprocessor-based Boards*, ITC'98: IEEE International Test Conference, Washington (USA), September 1998

[BRIM98] A. Benso, M. Rebaudengo, L. Impagliazzo, P. Marmo, *Fault List Collapsing for Fault Injection Experiments*, Annual Reliability and Maintainability Symposium, January 1998, Anaheim, California, USA, 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

[CTIy97] M.C. Hsueh, T. Tsai, R.K. Iyer, *Fault Injection Techniques and Tools*, IEEE Computer, Aprile 1997, pp. 75-82

[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, 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

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

[Lovr95] T. Lovric, *Processor Fault Simulation with ProFI*, European Simulation Symposium ESS95, 1995, pp. 353-357

[Moto92a] Motorola Inc., M68000 Family Integrated Development Platform (IDP), 1992

[Moto92b] Motorola Inc., M68000 Family Programmer's Reference Manual - M68000PM/AD, 1992

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

[PVBW88] D. Powell, P. Verissimo, G. Bonn, F. Waeselynck, D. Seaton, *The Delta-4 Approach to Dependability in Open Distributed Computing Systems*, Proc. FTCS-18, Tokyo (J), 1988, pp. 246-251

[SCMC96] J. G. Silva, J. Carreira, H. Madeira, D. Costa, F. Moreira, *Experimental Assessment of Parallel Systems*, Proc. FTCS-26, Sendaj (J), 1996, pp. 415-424

[RSeo99] M. Rebaudengo, M. Sonza Reorda, *Evaluating the Fault Tolerance Capabilities of Embedded Systems via BDM*, VTS'99: 17th IEEE VLSI Test Symposium, Dana Point (CA), 1999

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

[YIGo93] L. T. Young, R. Iyer, K. K. Goswami, *A Hybrid Monitor Assisted Fault injection Experiment*, Proc. DCCA-3, 1993, pp. 163-174