Comparison of Physical and Software-Implemented Fault Injection Techniques

Jean Arlat, *Member*, *IEEE*, Yves Crouzet, Johan Karlsson, *Member*, *IEEE*, Peter Folkesson, *Member*, *IEEE*, Emmerich Fuchs, *Member*, *IEEE Computer Society*, and Günther H. Leber, *Member*, *IEEE*

Abstract—This paper addresses the issue of characterizing the respective impact of fault injection techniques. Three physical techniques and one software-implemented technique that have been used to assess the fault tolerance features of the MARS fault-tolerant distributed real-time system are compared and analyzed. After a short summary of the fault tolerance features of the MARS architecture and especially of the error detection mechanisms that were used to compare the erroneous behaviors induced by the fault injection techniques considered, we describe the common distributed testbed and test scenario implemented to perform a coherent set of fault injection campaigns. The main features of the four fault injection techniques considered are then briefly described and the results obtained are finally presented and discussed. Emphasis is put on the analysis of the specific impact and merit of each injection technique.

Index Terms—Fault injection techniques, experimental assessment, fault-tolerant computing, error detection coverage.

1 Introduction

The dependability assessment of a fault-tolerant computer system is a complex task that requires the use of different levels of evaluation approaches and related tools. In complement to other possible approaches such as proving or analytical modeling whose applicability and accuracy are significantly restricted in the case of complex systems, *fault injection* has long been recognized to be particularly attractive and useful. Indeed, by speeding up the occurrence of errors and failures, fault injection is, in fact, a method for *testing* the fault tolerance mechanisms with respect to a specific set of inputs they are meant to cope with: *the faults*. Fault injection can be applied either on a simulation model of the target fault-tolerant system or on a hardware-and-software implementation (e.g., see [1], [2], [3], [4]).

Clearly, simulation-based fault injection is desirable as it can provide early checks in the design process of fault tolerance mechanisms (e.g., see [5]). Nevertheless, it is worth noting that fault injection on a prototype featuring the actual interactions between the hardware and software dimensions of the fault tolerance mechanisms supplies a more realistic and necessary complement to validate their implementation in a fault-tolerant system.

- *J. Arlat and Y. Crouzet are with LAAS-CNRS, 7, Avenue du Colonel Roche, 31077 Toulouse Cedex 4, France. E-mail: {Jean.Arlat, Yves.Crouzet}@laas.fr.*
- *J. Karlsson and P. Folkesson are with the Department of Computer Engineering, Chalmers University of Technology, SE-412 96 Goteborg, Sweden. E-mail: {johan, peterf}@ce.chalmers.se.*
- *E. Fuchs was with the Vienna University of Technology. He is now with DECOMSYS-Dependable Computer Systems, Stumpergasse 48/14, A-1060 Wien, Austria. E-mail: fuchs@decomsys.com.*
- *G.H. Leber was with the Vienna University of Technology. He is now with Adcon Telemetry AG, Inkustraße 24, A-3400 Klosterneuburg, Austria. E-mail: guenther.leber@ieee.org.*

Manuscript received 14 Feb. 2000; revised 12 Feb. 2002; accepted 24 Feb. 2003.
For information on obtaining reprints of this article, please send e-mail to: tc@computer.org, and reference IEEECS Log Number 111469.

Initially, most studies related to the application of fault injection on a prototype of a fault-tolerant system relied on physical fault injection (ΦFI, for short), i.e., the introduction of faults through the hardware layer of the target system (e.g., see [6]). A trend favoring the injection of errors through the software layer for simulating physical faults (i.e., software-implemented fault injection—SWIFI for short) has emerged (e.g., see [7], [8], [9], [10], [11]). Such an approach facilitates the application of fault injection by overcoming several problems associated with ΦFI techniques (such as controllability, repeatability, etc.). Moreover, recent studies have shown that SWIFI was also able to emulate some types of software faults (e.g., see [12]).

Nevertheless, in spite of the difficulties in developing support environments and conducting experiments, ΦFI techniques enable real faults to be injected in a very close representation of the target system especially without any alteration to the software being executed. The large body of works concerning ΦFI used widely different techniques and/or were applied to distinct target systems. This significantly hampers the possibility to identify the difficulties/benefits associated with each fault injection technique and to analyze the results obtained.

Thus, more experimental work is needed to better establish the relationship and differences between the fault injection techniques that are available to help the designers in assessing the dependability and fault tolerance properties of a computer system. In particular, one key concern that is often related to fault injection-based experiments is usually termed *fault representativeness*, i.e., the plausibility of the supported fault model with respect to actual faults. In this paper, we advocate that the study of the impact and consequences of an injected fault (i.e., the error propagated) offers a more pragmatic and sensible means to address the representativeness issue. Accordingly, we distinguish between two categories of approaches, depending on whether the analysis concerns the erroneous behaviors provoked by

1) some specific fault injection technique with respect to a set of real faults or 2) the application of several fault injection techniques (most of the time in previous studies only two techniques were considered).

Clearly, in principle, the first approach is more desirable and convincing for assessing the accuracy of the behaviors induced by the fault injection techniques. However, it is not always easy to gather a large amount of objective data on behaviors caused by real faults to support such an analysis. Typical examples of related studies include the comparison of the impact of real faults with respect to 1) extensive data and code corruptions [13], 2) elementary source code mutations [14], or 3) application of the SWIFI technique [12].

For what concerns the second approach, the comparison of several fault injection techniques provides only an indirect means for assessing their representativeness; nevertheless, such an approach is well suited to obtaining extensive error data sets from which useful insights can be derived. Should the experiments using different fault injection techniques lead to similar behaviors, then the techniques can be considered as "equivalent" and, thus, the one that exhibits the most suitable practical properties (e.g., reachability, controllability, reproducibility, intrusiveness, etc.) should be preferred. However, if different behaviors are observed, then the techniques are rather complementary. Such an insight is very much helpful in light of the work devoted to developing dependability benchmarks (e.g., see [11], [15], [16]),[1] in particular to substantiate which kind of relevant "faultload" should be considered for such benchmarks. Among the related studies, we would like to refer to the works reported in [10], [17], [18], [19], [20], [21], [22] that addressed most of the currently available fault injection techniques, including simulation-based techniques, ΦFI techniques, SWIFI techniques, and, also, the recently introduced scan chain-implemented fault injection technique, e.g., see [23], that builds upon the testability-support capabilities featured by many modern VLSI devices.

These studies showed that some fault injection techniques matched some real faults rather well and also that some were found to be quite equivalent, while others were identified as rather complementary. Accordingly, more experimental work and related analyses are needed to better understand the underlying error creation and propagation mechanisms.

This paper is intended to contribute to this effort along the lines of the second approach described earlier, by studying whether the application of four distinct fault injection techniques had the same impact on a specific prototype of a fault-tolerant real-time system. Three ΦFI techniques, namely, heavy-ion radiation, pin-level injection, and electromagnetic interferences, as well as a preruntime SWIFI technique (at the machine code level) were considered. In each case, the target fault-tolerant system was a prototype of the MARS (MAintainable Real-time System) distributed architecture developed at the Vienna University of Technology [24]. It is worth noting that this conceptual prototype architecture has evolved to become the Time Triggered Protocol and Architecture (also known as TTP and TTA), e.g., see [25].

The initial motivation for our work was to assess the *coverage* of the "fail-silent" assumption [26] and also to evaluate the respective efficiency (i.e., the detection *coverage*) of the various built-in error detection mechanisms (EDMs) aimed at supporting the fail silence property for the distributed computing nodes of the MARS architecture. The results obtained were previously reported in [27], for what concerns the experiments related to the three ΦFI techniques, and in [28] for the SWIFI experiments.

It is worth noting that, in order to carry out all the fault injection experiments on a consistent basis, we used the same distributed testbed architecture featuring five MARS nodes and a common test scenario. Accordingly, it was easy to extend our analyses toward the comparison of the erroneous behaviors provoked by the considered fault injection techniques. A preliminary comparative study of the coverage provided by the EDMs and of *fail silence* property achieved with respect to the ΦFI and SWIFI techniques was reported in [29]. This paper significantly extends these results and provides some insights to help understand 1) the differences between the errors provoked by the ΦFI techniques and 2) to what extent SWIFI can simulate the consequences of faults injected by the physical techniques.

To the best of our knowledge, this study is rather unique, both in providing such a comprehensive comparison of several fault injection techniques and in relying on a well-controlled experimental context that allowed for drawing meaningful comparisons. Indeed, the assessment of the fault injection techniques is supported by using the EDMs in a MARS node as "observers" to characterize the erroneous behaviors induced by the faults injected by the techniques considered. The remaining part of this paper is composed of seven sections. Section 2 highlights the fault tolerance features of MARS, focusing on the EDMs built in each MARS node. Section 3 then presents the overall framework supporting the experimental assessment. It describes the common test scenario and testbed architecture being used for carrying out the fault injection experiments, as well as the failure predicates defined for characterizing the behavior of the target system in the presence of injected faults. Section 4 briefly describes the fault injection techniques considered. Some major experimental results concerning the target fault-tolerant system are presented and discussed in Section 5. Section 6 focuses on the analysis on the respective impact of the fault injection techniques considered. Section 7 complements this analysis by considering additional properties that also characterize the application of the fault injection techniques. Finally, Section 8 concludes the paper.

2 The MARS Architecture and Error Detection Mechanisms

This section summarizes the main fault tolerance features of the MARS architecture [24]. Fault tolerance issues at

1. Recent related efforts also include the SIG on Dependability Benchmarking established by IFIP WG 10.4 (http://www.dependability.org/wg10.4/SIGDeB) and the European Project on Dependability Benchmarking—DBench-Project IST 2000-25425 (http://www.laas.fr/dbench).

system-level are discussed first, then the structure of a special-purpose processing node designed to support these features in an optimal way is briefly described. Finally, special attention is paid to the identification and characterization of the error detection mechanisms (EDMs) built-in into a MARS node.

2.1 Fault Tolerance

Fault tolerance in MARS is based on "fail-silent" nodes operating in active redundancy and on sending duplicate messages on two redundant real-time buses. *Fail silence* is intended to describe the behavior of a computer that fails "cleanly" by just stopping to send messages in case a failure occurs [30]. Up to three processing nodes can execute identical software, thus forming a Fault-Tolerant Unit (FTU).

To achieve a deterministic timing behavior even in the presence of faults, the MARS system uses active redundancy for all processing and communication activities: Each process is executed simultaneously at all nodes of an FTU and each message is transmitted quasi-simultaneously on each of the broadcast channels. Due to the fail silence property, the results of all three nodes of an FTU are assumed to be correct and may be used interchangeably. Since only two nodes are needed to tolerate a single failure of a fail-silent node (i.e., the loss of a message), the optional third node, the shadow node, does not transmit any message on the real-time network as long as both active nodes are operational. Only if an active node fails does the shadow node immediately start to transmit its results, thus restoring the initial degree of redundancy. A precise global time is maintained by a distributed fault-tolerant clock synchronization algorithm [24].

MARS uses a two-layered mechanism for fault tolerance. Due to the fail silence assumption supporting the design of the node, the top layer (system layer) need not care about erroneous data; it only has to provide enough redundancy to tolerate (silent) failures of parts of the system. Indeed, the bottom layer (node layer) is responsible for error detection and error confinement (i.e., it ensures the fail silence property of the node: After the detection of an error, a reset of the node is performed). In the context of this paper, a MARS node is said to be fail-silent if it only sends: 1) correct messages, 2) no messages, or 3) detectably wrong messages, which can be discarded by each nonfaulty receiver. An additional feature in the MARS architecture is that a node is only allowed to send a message at fixed time intervals according to a TDMA media access strategy.

2.2 Structure of the Processing Node

This study uses a single-board implementation of the MARS nodes. Each node consists of two independent processing units: the application unit and the communication unit (Fig. 1). Each unit is based on a 68070 CPU [31], that features a memory management unit (MMU). The application unit also contains a dynamic RAM and two bidirectional FIFOs, one of which serves as an interface to external add-on hardware, the other one connecting the application unit to the communication unit. Additional hardware for the communication unit is comprised of a Static RAM, two ethernet controllers (LANCEs), each coupled to a Clock Synchronization Unit (CSU) for maintaining a global time base, and a Time Slice Controller (TSC) for controlling access to the system bus.

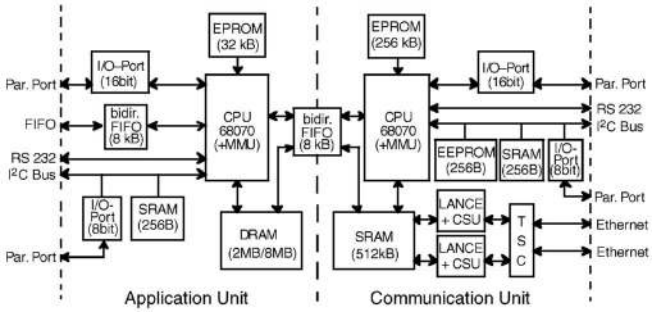

Fig. 1. Block diagram of the processor board.

Whenever an error is detected, the subsequent error processing activity of the node is to save the error information into nonvolatile memory and then turn itself off. Upon restart, the node writes its previously saved error information to two serial ports (one for each unit), from where it can be read for the purpose of diagnosis. This feature was exploited in the context of this study to precisely monitor and characterize the consequence of the injected faults. More details on MARS features and on the architecture of the processing nodes can be found in [24].

Three levels of error detection mechanisms (EDMs) are implemented in the MARS nodes: 1) the hardware EDMs, 2) the system software EDMs implemented in the operating system [24] and support software (i.e., the Modula/R compiler) [32], and 3) the application-level (end-to-end) EDMs at the highest level. They are respectively described in the following paragraphs.

2.3 Hardware Error Detection Mechanisms

Whenever an error is detected by one of the hardware EDMs, in general, an *exception* is raised and the two CPUs will then wait for a reset issued by a watchdog timer. This watchdog timer is the only device that may cause a reset of all devices, including the CPUs.

Two main categories of hardware EDMs can be distinguished: the built-in mechanisms of the CPUs and those provided by special hardware on the processing board. In addition, faults can also trigger "unexpected" exceptions (i.e., neither the EDMs built into the CPUs nor the mechanisms provided by special hardware are mapped to these exceptions).

The EDMs built into the CPUs are: bus error, address error, illegal op-code, privilege violation, zero-divide, stack format error, noninitialized vector interrupt, and spurious interrupt. These errors cause the processor to jump to the appropriate exception handling routines, which save the error state to the nonvolatile memory and then reset the node.

The following errors are detected by mechanisms implemented by special hardware on the node: silent shutdown of the CPU of the communication unit, power failure, parity error, FIFO over/underflow, access to physically nonexisting memory, write access to the real-time network at an illegal point in time (monitored by the TSC), error of an external device, and error of the other unit. We globally call these "NMI mechanisms" as they raise a

Non-Maskable Interrupt (a specific exception number) when an error is detected. An NMI leads to the same error handling as EDMs built into the CPUs and can only be cleared by resetting the node, which is carried out by the watchdog timer.

2.4 System Software Error Detection Mechanisms

These mechanisms consist of EDMs implemented by the operating system or special system tasks; they include:

- assertions built into the operating system (OS), such as integrity checks on data or processing time overflow,
- mechanisms inserted by the compiler (i.e., Compiler Generated Run-Time Assertions—CGRTA) to implement concurrent checks, such as value range overflow of a variable and loop iteration bound overflow.

When an error is detected by any of these mechanisms, a "trap" instruction is executed that leads to a node reset.

2.5 End-to-End Error Detection Mechanisms

These mechanisms include end-to-end checksums for message data and multiple (basically, double) execution of tasks.

The end-to-end checksums are used to detect the mutilation of message data exchanged between two nodes of an FTU and are therefore used by the receiving task for extending the fail silence property of the MARS nodes.

Double execution of tasks in time redundancy can detect errors caused by transient faults that cause different output data of the two instances of the task. Combined with the concept of message checksums, task execution in time redundancy forms the highest level in the hierarchy of the error detection mechanisms. These mechanisms also trigger the execution of a trap instruction, which causes a reset of the node.

3 Overview of the Experimental Framework

In this section, we first present the common testbed set-up and workload implemented at all sites for carrying out the experiments. Then, we precisely define the failure predicates considered during the fault injection experiments.

3.1 The Experimental Testbed and Workload

As depicted in Fig. 2, the common distributed testbed that is supporting the fault injection experiments features five MARS nodes.

The node under test (NUT, for short) is the node subject to the injection of a fault during each experiment run. Another node (*golden* node) serves as a reference and a third node (*comparator* node) is used to compare the messages sent by the two previous nodes. When a discrepancy is observed by the comparator node (fail silence violation) or the NUT detects an error, the NUT is declared to be failed and then shut down by the comparator node to clear all error conditions for the subsequent experiment run. After some time, power is reinstalled and the NUT is reloaded for the next experiment run. The *data generation* node simulates the data corresponding to the real-time application that is being used to activate the NUT and the golden node during each fault injection experiment.

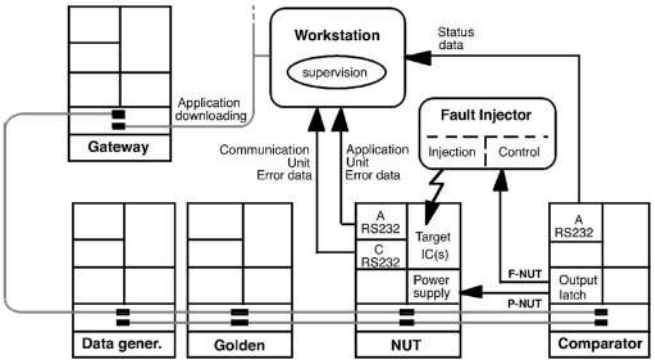

Fig. 2. The testbed architecture featuring five MARS nodes.

The application is taken from the rolling ball demonstration [33]: A ball is kept rolling along a circular path on a tiltable plane by controlling the two horizontal axes of the plane by servo motors and observing the position of the ball with a video camera. However, the tiltable plane and the camera are not present in the set-up used in the fault injection experiments; instead, the data from the camera is simulated by a data generation task running on the *data generation* node. The task provides the nominal and actual values of the position, speed, and acceleration of the ball.

A fifth node is included that serves as a *gateway* between a local area network (LAN) and the MARS network. It is required for loading the entire application and for reloading the NUT. A host computer (Unix workstation) connected to the LAN is used for supervising the experiments, i.e., reloading failed nodes and collecting data from each experiment run for further analysis.

Fig. 2 also depicts the specific interactions with the ΦFI devices. The experiments are managed by the workstation and controlled by the comparator node. When the comparator node detects an error, it reports the error type to the workstation and turns off the power to the NUT with the signal P-NUT. Signal F-NUT is used to discontinue fault injection.[2] Then, the NUT is powered-up again and restarted. Upon restart, the memorized error data is sent to the workstation via two serial lines (one for each processing unit).[3] Once the NUT has been restarted, the workstation immediately initiates the downloading of the application via the gateway node. When the application has been restarted, the comparator node enables fault injection (signal F-NUT) and a new experiment run begins.

Finally, it is worth noting that the experimental set-up is based on the assumption that the nodes are *replica determinate* (both in value and in time domains), i.e., if provided with the same input data, replicated nodes deliver identical outputs in an identical order within a specified time interval. In particular, extensive runs without fault injection of the rolling-ball target application have demonstrated that the MARS prototype architecture supported

2. Such a direct control on the injected fault is not possible in the case of the software-implemented fault injection technique used (see Section 4.4).

3. If the error was not detected by the NUT itself, then the node has no error information available and sends only a status message.

TABLE 1
The Basic Predicates

Cold Start (*CS*)	Cold start (power on) of the NUT is performed after every experiment run, except when a system failure occurred.
Warm Start (*WS*)	Warm start (reset) of the NUT caused by the detection of i) an error by the node's EDMs (*Internal WS*) or, ii) an incoming or outgoing link failure by means of the top layer of the fault tolerance mechanism, i.e., the membership protocol [35] (*External WS*).
Message Loss (*ML*)	One message (or more) from NUT was lost (i.e., not received by the comparator node).
Message Mismatch (*MM*)	Reception by the comparator node of differing messages from golden node and NUT.
System Failure (*SF*)	Failure of either the golden, data generation, or comparator nodes.

this property. Besides its interest for handling replicated entities in real-time fault-tolerant systems (e.g., see [34]), such a feature proved very useful in the context of the fault injection experiments so that the faulty behaviors between the testbed instances used for supporting each fault injection technique could be meaningfully compared.

3.2 The Failure Predicates

Four failure types can be distinguished for the NUT:

1. The EDMs detect an error and the node stops sending messages on the MARS bus; then, the node stores the error condition into a nonvolatile memory and resets itself by means of the watchdog timer.
2. The node fails to deliver the expected application message(s) for one or several application cycles, but no error is detected by the EDMs.
3. The node delivers a syntactically correct message with erroneous content. This is a fail silence violation in the value domain, which is recognized as a mismatch between the messages sent by the NUT and the golden node.
4. The node sends a message at an illegal point in time and thus disturbs the traffic on the MARS bus. This is a fail silence violation in the time domain.

On every restart, the NUT writes its previously saved error data, if available (i.e., if an error was detected by the EDMs) and data about its state to two serial ports, where it can be read and stored for further processing. From these data, five predicates (events) can be derived (Table 1).

The *Cold Start (CS)* predicate characterizes the end of each data set. The other four predicates characterize four failure types. The assertion (occurrence) of the *Warm Start (WS)* predicate in the data corresponds to the normal case when the node under test detects the error (failure type 1). The assertion of *ML* corresponds to a *Message Loss* (failure type 2); this behavior is not a fail silence violation because no erroneous data is sent, but it cannot be regarded as normal operation. Irrespective of the other events, the assertion of a *Message Mismatch (MM)* (failure type 3) corresponds to a fail silence violation (in the data domain). There are two ways in which a *System Failure* (*SF*) may occur: 1) A fail silence violation in the time domain (failure type 4) affects the operation of the other nodes, or 2) another node than NUT experiences a real hardware failure during the experiments. Although, no *SF*-type failures were observed in the conducted experiments, this failure event is described for the sake of completeness.

Given these failure types, the number of fail silence (*FS*) violations can be counted as:

$$\#\text{FS Viol.} = \#\text{Exp. MM} + \#\text{Exp. SF},$$

where # *Exp. X* counts the number of experiments where an *X*-type failure was diagnosed (i.e., predicate *X* was asserted).

4 The Fault Injection Techniques

In this section, we briefly present the main features of the four fault injection techniques applied for the experimental assessment of the MARS system.

4.1 Heavy-Ion Radiation

The fault injection experiments with heavy-ion radiation (HI, for short) were carried out at Chalmers University of Technology in Göteborg, Sweden. Heavy-ion radiation from a Californium-252 source can be used to inject single event upsets, i.e., bit-flips at internal locations in integrated circuits (ICs) using a miniature vacuum chamber. Fig. 3 depicts the cross-sectional view of the miniature vacuum chamber. The pins of the target IC are extended through the bottom plate of the vacuum chamber so that the chamber with the circuit can be directly plugged into the socket of the circuit under test. The vacuum chamber contains an electrically controlled shutter, which is used to shield the circuit under test from radiation during bootstrapping.

A major feature of the heavy-ion fault injection technique is that faults can be injected into VLSI circuits at locations which are difficult (and mostly impossible) to reach by other techniques. The transient faults produced are also reasonably well spread at random locations within an IC as there are many sensitive memory elements in most VLSI circuits. As device feature sizes of integrated circuits are shrinking, radiation induced bit-flips, also known as soft errors, are becoming an increasingly important source of

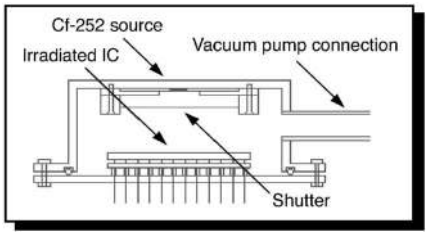

Fig. 3. Cross-sectional view of the miniature vacuum chamber.

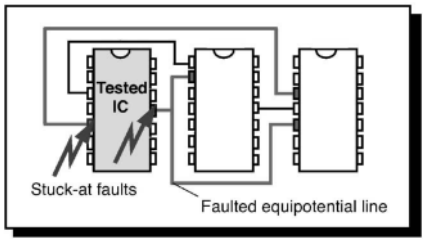

Fig. 4. Principle of pin-forcing fault injection.

failures in computer systems, e.g., see [36], [37]. While soft errors are caused mainly by heavy ions in space, at ground level and airplane flight altitudes, they are instead caused by atmospheric neutrons [38].

Although the heavy-ions emitted from Cf-252 do not provide a perfect imitation of the impact of either heavy ions in space or neutron radiation on earth (e.g., with respect to the ratio between multiple and single bit upsets), the method provides a practical approach to evaluating the effectiveness of error detection and recovery mechanisms with respect to soft errors. Exposing circuits to neutron radiation is less practical since it involves placing the tested system in a room with concrete shielding [39].

For the 68070 CPU, the heavy-ions from Cf-252 mainly provoke single bit upsets; the percentage of multiple bit errors induced in the main registers was found to be less than 1 percent in the experiments reported in [40]. The heavy-ion method has been previously used to evaluate several hardware and software-implemented error detection mechanisms for the MC6809E microprocessor. A comprehensive description of these experiments using the heavy-ion fault injection technique is given in [2].

4.2 Pin-Level Injection

Pin-level fault injection, i.e., the injection of faults directly on the pins of the ICs of a prototype, probably was the most widely applied physical fault injection technique. Some flexible tools supporting general features have been developed (e.g., see the test facility used on the MAFT system [41], MESSALINE [1], RIFLE [42], or AFIT [43]).

The experiments with the pin-level fault injection technique were conducted at LAAS-CNRS, in Toulouse, France, using MESSALINE. Fig. 4 depicts the principle of the pin-forcing technique (PF, for short) that was used. In the case of pin-forcing, the injected fault is directly applied on the pin(s) of the target IC.

It is noteworthy that the pins of the ICs connected, by means of an equipotential line, to an injected pin are faulted as well. Accordingly, to simplify the accessibility to the pins of the microprocessor, the target ICs were mainly the buffer ICs directly connected to it. The supported fault models include temporary stuck-at faults affecting single or multiple pins. Indeed, temporary faults injected on the pins of the ICs can simulate the consequences of internal faults on the pins of the faulted IC(s).

The tool already contributed to the experimental assessment of two fault-tolerant systems: 1) for testing the diagnosis features of a computerized railway interlocking system and 2) for evaluating the fail silence property of the Delta-4 fault-tolerant architecture [1], [30].

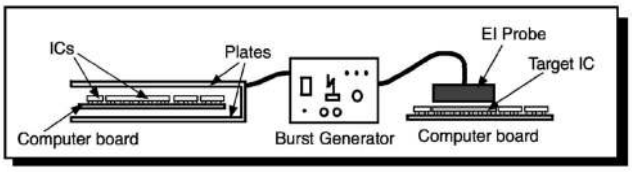

Fig. 5. Application of electromagnetic interferences.

4.3 Electromagnetic Interferences

Electromagnetic interferences (EI) are common disturbances in automotive vehicles, trains, airplanes, or industrial plants. Such a technique is widely used to stress digital equipment.

The EI experiments were carried out at the Technical University of Vienna, Austria. Thanks to the use of a commercial burst generator, this technique is easy to implement. Two different forms of application of this technique were considered (Fig. 5).

In the first form, the single computer board of the NUT was mounted between two metal plates connected to the burst generator. In this way, the entire node was affected by the generated bursts. Because the Ethernet transceivers turned out to be more sensitive to the bursts than the node under test itself, a second configuration was set up which used a special probe that was directly placed on top of the target circuit. In this way, the generated bursts affected only the target circuit (and some other circuits located near the probe).

4.4 Software-Implemented Fault Injection

Software-implemented fault injection (SWIFI) provides a low cost and easy-to-control alternative to the three physical fault injection techniques previously described that require special hardware instrumentation and interfaces to the target system. SWIFI is usually achieved by changing the contents of memory or registers based on specified fault models to emulate the consequences of hardware faults or to inject software faults (e.g., see [9]).

For these experiments, an alternative approach was selected that injected the faults at preruntime at the machine code level and loaded the mutilated application (code segment or data segment) to the target system afterward. Three main reasons led us to select such an approach [28]:

1. The intrusiveness is reduced to a minimum since faults are injected only into the application software (no additional code, which could probably alter the behavior of the application software, is needed, i.e., fault injection is transparent to the application),
2. Fault injection at the machine code level is capable of injecting faults which cannot be injected at higher levels by using source code mutations (e.g., see [44], [45]).
3. Preruntime injection smoothly integrates with the application development process because applications are developed, configured, allocated, and scheduled offline on a host computer and loaded onto the target system afterward.

SWIFI experiments started at the Vienna University of Technology, Austria, and continued at the Research and

TABLE 2
Experimental Configurations

Configuration #	Message Checksum	Execution	Acronym
1	No	Single	NOAM
2	Yes	Single	SEMC
3	Yes	Double	DEMC
4	Yes	Triple	TEMC

TABLE 3
Explanation of the Acronyms Used to Describe the Results

Acronym and Entry	Explanation
CPU	CPU-internal EDMs
UEE	Unexpected exceptions (i.e., without dedicated error handler)
NMI	Non-maskable interrupts generated by additional circuits of the NUT
OS	Checks and assertions coded in the operating system software
CGRTA	Compiler generated run-time assertions
Double exec.	Time redundant (Double) execution of application tasks
Checksum	Checksum of message exchanged
Other unit	The error information was provided by the other unit (not the faulted unit) of the NUT
No error info.	None of the two units of the NUT provided valid error information
Triple execution	Time redundant (Triple) execution of application tasks*
IM Loss	Intermittent loss of application message◊
No Reply	The injected fault(s) had no observable effect◊
Fail silence violations	Mismatch between the NUT and the golden node (as observed by the comparator node)

* Heavy-ion injection, only ◊SWIFI, only

Technology Institute of Daimler Benz AG (now Daimler-Chrysler) in Berlin, Germany [46].

The way the fault injection experiments are conducted differs slightly from the ΦFI experiments for which faults are injected until the injected node (NUT) fails. Indeed, this is not a feasible solution for SWIFI: Faults are likely to be overwritten before being activated or to be injected at locations that are not executed. In such cases, the NUT would continue operation infinitely. Accordingly, a time-out mechanism has been implemented in order to shut down the NUT after a prespecified time interval.

5 Results Obtained with Each Fault Injection Technique

Three combinations of the end-to-end EDMs were used for the four fault injection techniques considered, which led to the following three experimental configurations (see also Table 2):

- NOAM: no application level mechanisms, i.e., single execution and no checksums,
- SEMC: single execution, message checksums,
- DEMC: double execution, message checksums.

In addition, a fourth configuration TEMC (triple execution, message checksums) was used in the heavy-ion experiments (see Section 5.1).

In the following paragraphs, we present in sequence the results obtained by the application of each technique. Then, these results are discussed in a subsequent paragraph. These results are further analyzed in the next section to support the analysis of the fault injection techniques. For the sake of readability of the results, Table 3 provides an overview of the acronyms and entries found in the subsequent result tables.

5.1 Heavy-Ion Radiation

Two circuits in the NUT were irradiated in separate experiments: the CPU of the application unit and the CPU of the communication unit. Because the irradiated ICs were CMOS circuits, they had to be protected from heavy-ion induced latch-up.[4] The triggering of a latch-up is indicated by a drastic increase in the current drawn by the circuit. To prevent latch-ups from causing permanent damage to the ICs, a special device was used to turn off the power to the ICs when the current exceeded a threshold value.

Table 4 shows the distribution of error detections among the various EDMs for each of the irradiated CPUs, and the four combinations given in Table 2 (see also for the explanations of the entries). Table 4a gathers the results for the unit that contained the fault injected circuit. Table 4b complements these results with the error data reported by the other (fault free) unit of the NUT; these data detail the percentage reported by the "Other unit" category of Table 4a.

The hardware EDMs, in particular the CPU mechanisms, detected most of the errors. This is not surprising since the faults were injected *into* the CPU. The proportion of errors detected by the hardware EDMs is larger for faults injected into the communication CPU than for faults injected into the application CPU. In particular, the coverage of the NMI EDMs is higher in the former case. Unexpected exceptions occur with a frequency of about 15 percent in all combinations.

4. A latch-up is the triggering of a parasitic four layer switch (npnp or pnpn) acting as a silicon controlled rectifer (SCR), which may destroy the circuit due to excessive heat dissipation.

TABLE 4
Results for Heavy-Ion Radiation: (a) Detection by the EDMs of the Unit to Which the Faulted ICs Belong; (b) Detection by the EDMs of the Other Unit (Detail of "Other Unit" in (a))

Error Detection Mechanisms		Application unit (AU) CPU irradiated								Communication unit (CU) CPU irradiated					
		NOAM		SEMC		DEMC		TEMC		NOAM		SEMC		DEMC	
		Errors	%	Errors	%	Errors	%	Errors	%	Errors	%	Errors	%	Errors	%
Hardware	CPU	3735	47.7%	1410	49.0%	4280	47.4%	2573	51.3%	1113	44.9%	1270	43.2%	1056	43.3%
	UEE	1173	15.0%	459	16.0%	1373	15.2%	736	14.7%	361	14.6%	416	14.1%	326	13.4%
	NMI	549	7.0%	173	6.0%	570	6.3%	286	5.7%	500	20.2%	578	19.6%	484	19.9%
	Subtotal	*5457*	*69.7%*	*2042*	*71.0%*	*6223*	*68.9%*	*3595*	*71.7%*	*1974*	*79.6%*	*2264*	*76.9%*	*1866*	*76.6%*
System software	OS	610	7.8%	222	7.7%	687	7.6%	273	5.4%	90	3.6%	144	4.9%	128	5.3%
	CGRTA	75	1.0%	3	0.1%	30	0.3%	37	0.7%	10	0.4%	7	0.2%	13	0.5%
	Subtotal	*685*	*8.8%*	*225*	*7.8%*	*717*	*7.9%*	*310*	*6.2%*	*100*	*4.0%*	*151*	*5.1%*	*141*	*5.8%*
End-to end	Double exec.	—	—	—	—	75	0.8%	56	1.1%	—	—	—	—	11	0.5%
	Checksum	—	—	70	2.4%	247	2.7%	231	4.6%	—	—	48	1.6%	75	3.1%
	Subtotal	—	—	*70*	*2.4%*	*322*	*3.6%*	*287*	*5.7%*	—	—	*48*	*1.6%*	*86*	*3.5%*
Other	Other unit	1095	14.0%	381	13.2%	1295	14.3%	566	11.3%	342	13.8%	407	13.8%	293	12.0%
	No error info.	402	5.1%	122	4.2%	431	4.8%	216	4.3%	62	2.5%	73	2.5%	51	2.1%
	Subtotal	*1497*	*19.1%*	*503*	*17.5%*	*1726*	*19.1%*	*782*	*15.6%*	*404*	*16.3%*	*480*	*16.3%*	*344*	*14.1%*
Triple execution		—	—	—	—	—	—	42	0.8%	—	—	—	—	—	—
Fail silence violations		186	2.4%	37	1.3%	48	0.5%	0	0%	1	<0.1%	0	0%	0	0%
Total number of errors		*7825*	*100%*	*2877*	*100%*	*9036*	*100%*	*5016*	*100%*	*2479*	*100%*	*2943*	*100%*	*2437*	*100%*

(a)

Error Detection Mechanisms		AU CPU irradiated (Detection by the CU)								CU CPU irradiated (Detection by the AU)					
		NOAM		SEMC		DEMC		TEMC		NOAM		SEMC		DEMC	
		Errors	%	Errors	%	Errors	%	Errors	%	Errors	%	Errors	%	Errors	%
Hardware	CPU	0	0%	0	0%	0	0%	0	0%	0	0%	0	0%	0	0%
	UEE	3	<0.1%	2	0.1%	0	0%	3	0.1%	0	0%	0	0%	1	<0.1%
	NMI	199	2.5%	58	2.0%	243	2.7%	103	2.1%	118	4.7%	147	5.0%	103	4.2%
	Subtotal	*202*	*2.6%*	*60*	*2.1%*	*243*	*2.7%*	*106*	*2.1%*	*118*	*4.7%*	*147*	*5.0%*	*104*	*4.3%*
System software	OS	893	11.4%	321	11.2%	1052	11.6%	460	9.2%	224	8.9%	260	8.8%	189	7.8%
	CGRTA	0	0%	0	0%	0	0%	0	0%	0	0%	0	0%	0	0%
	Subtotal	*893*	*11.4%*	*321*	*11.2%*	*1052*	*11.6%*	*460*	*0.2%*	*224*	*8.9%*	*260*	*8.8%*	*189*	*7.8%*
End-to end	Double exec.	—	—	—	—	0	0%	0	0%	—	—	—	—	0	0%
	Checksum	—	—	0	0%	0	0%	0	0%	—	—	0	0%	0	0%
	Subtotal	—	—	*0*	*0%*	*0*	*0%*	*0*	*0%*	—	—	*0*	*0%*	*0*	*0%*

(b)

Errors detected by the Operating System (OS) mechanisms dominate the "System software" EDMs, while detection by the message checksums dominate the "End-to-end" EDMs.

Percentages for fail silence violations were between 2.4 percent and 0.5 percent for the NOAM, SEMC, and DEMC combinations when faults were injected into the application CPU. As expected, the number of fail silence violations is lower for SEMC than for NOAM and even lower for DEMC. Moreover, when faults were injected into the communication CPU, one single fail silence violation was observed (for NOAM).

The percentage of fail silence violations (0.5 percent) observed for the DEMC combination was unexpected. In principle, all effects of transient faults should be masked by the double execution of tasks. One hypothesis for explaining these violations is that, despite the specific protection device used, an undetected latch-up caused the same incorrect result to be produced by both executions of the control task. To further investigate this hypothesis, experiments were carried out with the TEMC combination that used a third time-redundant execution of the control task which was provided with fixed input data for which the results were known. This made it possible to detect errors by comparing the produced results with the correct results. This mechanism, which can be viewed as an online test program, would detect any semipermanent fault such as the one suggested by the latch-up hypothesis. The results show that no fail silence violations occurred for the TEMC combination. As Table 4a shows, 0.8 percent of the errors were detected by the third execution of the control task. This result comes in support of the latch-up hypothesis. However, our experimental set-up does not provide sufficient observability to fully prove the latch-up hypothesis. In principle, the absence of fail silence violations could also be an effect of the change of the software configuration caused by the switch from DEMC to TEMC and the errors detected by the third execution may have been caused by regular transients. Verification of the latch-up hypothesis would require the use of a logic analyzer so that the program flow and behavior of the microprocessor could be studied in detail.

The OS and NMI EDMs dominate the detections made by the other unit of the NUT. The communication between the two units is done via two FIFO buffers (see Fig. 1) and nearly all of these detections are made by EDMs signaling empty FIFO. (An empty FIFO can be detected both by the executive software and the special NMI mechanism.)

5.2 Pin-Level Injection

The forcing technique was used for the experiments carried out on the MARS system. The main characteristics of the injected faults are listed hereafter:

TABLE 5
Results for Pin-Forcing Injection: (a) Detection by the EDMs of the Unit to Which the Faulted ICs Belong; (b) Detection by the EDMs of the Other Unit (Detail of "Other Unit" Entry in (a))

Error Detection Mechanisms		ICs of the application unit (AU)						ICs of the communication unit (CU)					
		NOAM		SEMC		DEMC		NOAM		SEMC		DEMC	
		Errors	%	Errors	%	Errors	%	Errors	%	Errors	%	Errors	%
Hardware	CPU	71	11.2%	53	9.0%	38	7.0%	37	6.9%	37	8.2%	20	3.9%
	UEE	48	7.6%	59	10.0%	41	7.6%	113	21.2%	73	16.2%	103	19.8%
	NMI	474	75.0%	430	73.0%	423	78.2%	265	49.7%	260	57.5%	263	50.7%
	Subtotal	*593*	*93.8%*	*542*	*92.0%*	*502*	*92.8%*	*415*	*77.9%*	*370*	*81.9%*	*386*	*74.4%*
System software	OS	6	0.9%	6	1.0%	7	1.3%	35	6.6%	21	4.6%	30	5.8%
	CGRTA	0	0%	1	0.2%	0	0%	0	0%	0	0%	0	0%
	Subtotal	*6*	*0.9%*	*7*	*1.2%*	*7*	*1.3%*	*35*	*6.6%*	*21*	*4.6%*	*30*	*5.8%*
End-to end	Double exec.	—	—	—	—	0	0%	—	—	—	—	0	0%
	Checksum	—	—	0	0%	0	0%	—	—	1	0.2%	5	1.0%
	Subtotal	—	—	*0*	*0%*	*0*	*0%*	—	—	*1*	*0.2%*	*5*	*1.0%*
Other	Other unit	1	0.2%	8	1.4%	2	0.4%	23	4.3%	17	3.8%	26	5.0%
	No error info.	32	5.1%	30	5.1%	30	5.5%	59	11.1%	43	9.5%	72	13.9%
	Subtotal	*33*	*5.2%*	*38*	*6.5%*	*32*	*5.9%*	*82*	*15.4%*	*60*	*13.3%*	*98*	*18.9%*
Fail silence violations		0	0%	2	0.3%	0	0%	1	0.2%	0	0%	0	0%
Total number of errors		*632*	*100%*	*589*	*100%*	*541*	*100%*	*533*	*100%*	*452*	*100%*	*519*	*100%*

(a)

Error Detection Mechanisms		ICs of the AU (Detection by the CU)						ICs of the CU (Detection by the AU)					
		NOAM		SEMC		DEMC		NOAM		SEMC		DEMC	
		Errors	%	Errors	%	Errors	%	Errors	%	Errors	%	Errors	%
Hardware	CPU	0	0%	0	0%	0	0%	0	0%	0	0%	0	0%
	UEE	0	0%	0	0%	0	0%	0	0%	0	0%	2	0.4%
	NMI	1	0.2%	7	1.2%	2	0.4%	23	4.3%	17	3.8%	24	4.6%
	Subtotal	*1*	*0.2%*	*7*	*1.2%*	*2*	*0.4%*	*23*	*4.3%*	*17*	*3.8%*	*26*	*5.0%*
System software	OS	0	0%	1	0.2%	0	0%	0	0%	0	0%	0	0%
	CGRTA	0	0%	0	0%	0	0%	0	0%	0	0%	0	0%
	Subtotal	*0*	*0%*	*1*	*0.2%*	*0*	*0%*	*0*	*0%*	*0*	*0%*	*0*	*0%*
End-to end	Double exec.	—	—	—	—	0	0%	—	—	—	—	0	0%
	Checksum	—	—	0	0%	0	0%	—	—	0	0%	0	0%
	Subtotal	—	—	*0*	*0%*	*0*	*0%*	—	—	*0*	*0%*	*0*	*0%*

(b)

- One single IC was fault injected at a time (the maximum number of pins faulted simultaneously—i.e., the multiplicity of the fault—being limited to $mx = 3$),
- Uniform distribution over all combinations of mx pins out of the n functional pins of the target IC was used to select the mx faulted pins,
- Stuck-at-0 and -1 fault models (all 0-1 combinations of mx pins were considered equally probable),
- To facilitate the comparison with the other techniques, both transient and intermittent (series of transients) faults were injected.

As a consequence of the application of the pin-forcing technique, it can be confidently considered that all pins of the ICs connected to an actually injected pin are equally faulted. Accordingly, to simplify the accessibility to the pins of the CPUs of the application and communication units, the target ICs were mainly buffer ICs connected to them. As a result, seven ICs (five on the application unit and two on the communication unit) were tested. These tests resulted in 3,266 error reports.

Table 5 shows the distribution of the errors detected by the various EDMs, together with their percentage of the total number of errors observed in each experimental configuration.

The results in Table 5a indicate a dominant proportion of detections by the hardware EDMs (more than 90 percent on the application unit side and 75 percent on the communication unit side). NMIs clearly dominate; however, in addition to CPU exceptions, a significant number of UEEs were also triggered. The difference between UEE and NMI for the application and communication units can be explained by the fact that not all ICs tested on the application unit are directly connected to the processor. In the "System software" category, the OS EDMs significantly dominate. Concerning the "End-to-end" level, the "Checksum" detections significantly dominate: No detections were triggered by the "Double execution" when this option was enabled. Only a limited number of fail silence violations were observed: two occurrences for the SEMC combination when injecting on the application unit and one occurrence for the NOAM combination when injection targeted the communication unit.

Table 5b shows that NMI EDMs also dominate the supplementary detections observed on the other unit. A significant difference is observed between the results of whether the injection affects the application unit or the communication unit; this may indicate that a larger proportion of errors was propagated to the application unit.

TABLE 6
Results for Electromagnetic Interferences: (a) Detection by the EDMs of the Unit to Which the Faulted ICs Belong; (b) Detection by the EDMs of the Other Unit (Detail of "Other Unit" Entry in (a))

Error Detection Mechanisms		Injection with antennas						Injection with probe only					
		NOAM(1)		SEMC(2)		DEMC(3)		NOAM(4)		SEMC(5)		DEMC(6)	
		Errors	%	Errors	%	Errors	%	Errors	%	Errors	%	Errors	%
Hardware	CPU	1195	72.0%	193	76.6%	137	2.2%	4933	99.4%	1692	98.1%	1911	99.2%
	UEE	11	0.7%	8	3.2%	9	0.2%	31	0.6%	17	1.0%	15	0.8%
	NMI	48	2.9%	18	7.1%	695	11.4%	0	0%	3	0.2%	0	0%
	Subtotal	*1254*	*75.6%*	*219*	*86.9%*	*841*	*13.8%*	*4964*	*100%*	*1712*	*99.3%*	*1926*	*100%*
System software	OS	110	6.6%	5	2.0%	5215	85.6%	0	0%	3	0.2%	0	0%
	CGRTA	5	0.3%	0	0%	1	<0.1%	0	0%	0	0%	0	0%
	Subtotal	*115*	*6.9%*	*5*	*2.0%*	*5216*	*85.6%*	*0*	*0%*	*3*	*0.2%*	*0*	*0%*
End-to end	Double exec.	–	–	–	–	9	0.2%	–	–	–	–	0	0%
	Checksum	–	–	1	0.4%	8	0.1%	–	–	1	<0.1%	0	0%
	Subtotal	-	-	*1*	*0.4%*	*17*	*0.3%*	-	-	*1*	*<0.1%*	*0*	*0%*
Other	Other unit	–	–	24	9.5%	6	0.1%	0	0%	6	0.3%	0	0%
	No error info.	271	16.3%	0	0%	13	0.2%	0	0%	2	0.1%	0	0%
	Subtotal	*271*	*16.3%*	*24*	*9.5%*	*19*	*0.3%*	*0*	*0%*	*8*	*0.4%*	*0*	*0%*
Fail silence violations		20	1.2%	3	1.2%	0	0%	0	0%	0	0%	0	0%
Total number of errors		*1660*	*100%*	*252*	*100%*	*6093*	*100%*	*4964*	*100%*	*1724*	*100%*	*1926*	*100%*

(a)

Error Detection Mechanisms		Injection with antennas						Injection with probe only					
		NOAM(1)		SEMC(2)		DEMC(3)		NOAM(4)		SEMC(5)		DEMC(6)	
		Errors	%	Errors	%	Errors	%	Errors	%	Errors	%	Errors	%
Hardware	CPU	–	–	0	0%	0	0%	0	0%	0	0%	0	0%
	UEE	–	–	0	0%	0	0%	0	0%	0	0%	0	0%
	NMI	–	–	0	0%	6	0.1%	0	0%	0	0%	0	0%
	Subtotal	–	–	*0*	*0%*	*6*	*0.1%*	*0*	*0%*	*0*	*0%*	*0*	*0%*
System software	OS	–	–	24	9.5%	0	0%	0	0%	6	0.3%	0	0%
	CGRTA	–	–	0	0%	0	0%	0	0%	0	0%	0	0%
	Subtotal	–	–	*24*	*9.5%*	*0*	*0%*	*0*	*0%*	*6*	*0.3%*	*0*	*0%*
End-to end	Double exec.	–	–	–	–	0	0%	–	–	–	–	0	0%
	Checksum	–	–	0	0%	0	0%	–	–	0	0%	0	0%
	Subtotal	–	–	*0*	*0%*	*0*	*0%*	–	–	*0*	*0%*	*0*	*0%*

(b)

5.3 Electromagnetic Interferences

Various fault injection campaigns were carried out with a variety of voltage levels, with negative or positive polarity of the bursts, and with a burst-frequency of 2.5 kHz and 10 kHz. A total number of more than 17,000 errors were observed during all campaigns conducted with the first method, i.e., when the computer board of the node under test was mounted between two plates, and more than 30,000 errors were observed using the special probe (see Section 4.3). Most of the campaigns were conducted with all application level EDMs enabled.

In the first campaign of Table 6 (identified as NOAM(1)), faults were injected into the communication unit using the two plates. Antenna wires were attached to the so-called LO-EPROM in order to disturb the address bus and the eight low order bits of the data bus. Bursts characterized by a frequency of 2.5 kHz, negative polarity, and a voltage of 230 V were injected. The second campaign (SEMC(2)) used the special probe, with antenna wires connected to the LO-EPROM in the application unit. In this case, the bursts were characterized by a frequency of 10 kHz, negative polarity and a voltage of 300 V. Campaign number three (DEMC(3)) used the two plates, the bursts had a frequency of 2.5 kHz, negative polarity, and voltage of 230 V. The wires were attached to the LO-EPROM of the application unit. Campaigns 4 to 6 were only using the special probe for coupling faults into the CPU of the application unit, i.e., the probe was mounted on top of the CPU and no wires were attached to any chip. The chosen frequency for the bursts was 10 kHz and negative polarity was used for all these experiments. We used a voltage of 290 V for campaigns 4 and 6, while a slightly higher voltage, 300 V, was used for campaign 5.

Due to the large number of campaigns made, only selected campaigns are presented in Table 6, which shows the distribution of the errors detected by the various EDMs as total numbers and as percentage. Table 6a shows the errors detected by the unit where fault injection was focused to; errors detected by the other unit of the NUT are detailed in Table 6b.

Campaigns 1 and 2 show similar results, although focus of fault injection was on different units of the NUT, the communication unit for the first and the application unit for the second. Most of the errors were detected by the hardware EDMs, where the CPU EDMs clearly dominate. For the "system software" EDMs, which only detected a small fraction of the errors, the OS category dominates. The relatively high amount of occurrences of the "No error info." category for campaign 1 partly results from the fact that, for this campaign, no information about the errors detected by the application unit is available because this is a result from early experiments, where only the outputs of the

TABLE 7
Results for Software-Implemented Fault Injection

Error Detection Mechanisms		Injection into the code segment						Injection into the data segment					
		NOAM		SEMC		DEMC		NOAM		SEMC		DEMC	
		Exper.	%	Exper.	%	Exper.	%	Exper.	%	Exper.	%	Exper.	%
Hardware	CPU	443	14.6%	769	25.4%	679	22.6%	0	0%	0	0%	0	0%
	UEE	1433	47.4%	440	14.5%	732	24.4%	1711	57.0%	88	2.5%	123	4.1%
	NMI	0	0%	0	0%	0	0%	0	0%	0	0%	0	0%
	Subtotal	1876	62.0%	1209	40.0%	1411	47.0%	1711	57.0%	88	2.5%	123	4.1%
System software	OS	3	0.1%	13	0.4%	12	0.4%	2	0.1%	0	0%	0	0%
	CGRTA	103	3.4%	113	3.7%	87	2.9%	0	0%	0	0%	0	0%
	Subtotal	106	3.5%	126	4.2%	99	3.3%	2	0.1%	0	0%	0	0%
End-to end	Double exec.	0	0%	0	0.0%	289	9.6%	0	0%	0	0%	626	20.9%
	Checksum	1	0.0%	211	7.0%	195	6.5%	0	0%	2378	68.0%	1787	59.5%
	Subtotal	1	0.0%	211	7.0%	484	16.1%	0	0%	2378	68.0%	2413	80.4%
Other	Other unit	1	0.0%	0	0%	0	0%	0	0%	2	0.1%	1	0.0%
	No error info.	8	0.3%	112	3.7%	0	0%	2	0.1%	0	0%	1	0.0%
	Subtotal	9	0.3%	112	3.7%	0	0%	2	0.1%	2	0.1%	2	0.1%
IM Loss		0	0%	21	0.7%	0	0%	0	0%	494	14.1%	0	0%
No Reply		728	24.1%	1090	36.0%	1006	33.5%	1277	42.6%	534	15.3%	464	15.5%
Fail silence violations		304	10.1%	256	8.5%	0	0%	8	0.3%	0	0%	0	0%
Total number of exp.		3024	100%	3025	100%	3000	100%	3000	100%	3496	100%	3002	100%

unit under test were recorded and, therefore, all errors that were detected by the application unit are counted as "No error info.".

A singular distribution of error reports was observed for campaign 3. There, the system software EDMs (especially OS) detected most of the errors. Most reports pointed out that a message which was required by the application was lost. Although both campaigns 1 and 3 used the "two metal plates" technique, the observed results are quite different. Conversely, while campaigns 1 and 2 had different EI conditions, the results observed are very similar. This observation and the unique feature of the results obtained during campaign 3 led us to consider them suspiciously. More generally, significantly different results were observed for similar conditions, e.g., slight changes in voltage levels. Thus, reproducibility appears to be problematic for EI experiments.

For campaigns 4 to 6, almost all of the errors were detected by the CPU EDMs. Only campaign 5 shows a small amount of errors detected by other EDMs than hardware EDMs. When looking at the results of campaigns 4 to 6 in more detail, we discovered that almost all of the detected errors were spurious interrupts detected by the CPU. Spurious interrupts are interrupts signaled to the processor, but the processor cannot find the source of the interrupt, i.e., the device having raised the interrupt. This shows that the interrupt lines of a processor are highly sensitive to EI.

For all campaigns, errors detected by the "Other unit" were only detected by the NMI EDMs and by the OS EDMs.

5.4 Software-Implemented Fault Injection

Both the code and data segments of the rolling-ball application software were targeted by the SWIFI technique.[5] Within each segment, the bit to be faulted was selected randomly to achieve a uniform distribution over the whole segment. To facilitate the comparison with the ΦFI techniques, we only consider here the single bit-flip experiments because they constitute a reasonable fault scenario for the comparison with these techniques (e.g., heavy-ion radiation generates, to a large extent, single bit-flips).

Table 7 shows the distribution of the errors detected by the various EDMs, together with their percentage of the total number of experiments observed in each experimental configuration.

Three comments need to be made prior to analyzing these results in detail:

1. The bit-flips affect the code and data segments that are processed by the application unit, so the injection implicitly focuses on the application unit. The fact that the percentages observed for the "Other unit" category (i.e., the error reports returned by the communication unit) are almost negligible indicates a limited propagation of the errors.
2. As opposed to ΦFI experiments, where the experimental protocol prevented such occurrences,[6] a significant ratio of SWIFI experiments (up to 42 percent in the case of data segment injection) led to nonsignificant experiments where no effect could be observed (these are categorized as "NoReply" in Table 7). Such a significant proportion is a common finding in other reported work on SWIFI (e.g., see [8], [9], [11], [16], [47]). While this proportion is decreased in the case data segment injection when application-level EDMs are enabled, the opposite observation is made in the case of injection into the code segment: This can be explained by the fact that application-level EDMS are introducing additional code for implementing the checks that result in an increase in the number of potential unused areas. Nevertheless, these results exemplify the real benefits that one can expect from applying such EDMs to protect with respect to data errors.

5. To carry out a fair comparison with the ΦFI experiments, only the results obtained while executing the rolling ball application are considered here (see [28] for more results on the SWIFI experiments).

6. Remember that the ΦFI fault injection experiments were carried out until the NUT failed. This is not a feasible solution for SWIFI (e.g., faults can be injected at locations that are not reached by the execution process or errors can be overwritten before being propagated).

3. Another singular behavior was the intermittent omission of application messages ("IM Loss" category). Such a singular behavior was quite significant in the case of data segment injection (14 percent)! Analysis of the cause was traced to the mutilation of the time-stamp information incorporated into the end-to-end CRC (see [29] for more details).

The first observation is that the behaviors highly depend on the type of segment targeted by fault injection. While the proportion of hardware error detections dominates in the case of code segment injection, this is no longer the case for data segment injection experiments: On the contrary, end-to-end EDMs significantly dominate. Moreover, especially in the NOAM configuration, the type of hardware EDMs exercised highly depend on the type of segment targeted by fault injection: While CPU EDMs were exercised frequently in the case of code segment injection, only UEEs have been observed for data segment injection.

Another interesting difference between the target segments concerns the system software EDMs: Indeed, as could be expected, these do not contribute (nearly at all) to the detection in the case of data segment injection. Also, occurrences of fail silence violations were significantly higher for code segment injection. However, in both cases, the utilization of end-to-end EDMs (especially, double execution[7]) in addition to hardware and system software EDMs proved useful to eliminate this risk.

5.5 Discussion

We synthesize here the results related to the evaluation of the efficiency of the various EDMs implemented in the MARS architecture. In particular, these results provide objective insights to the designers in getting confidence in the way the fail silence property of the computing nodes has been achieved. The analysis focuses on the complementarity of the EDMs in contributing to the fail silence property. The detailed analysis of the different erroneous behaviors provoked by the fault injection techniques considered is presented in Section 6.

Most ΦFI fault injection campaigns show that the hardware EDMs significantly detect most of the errors.[8] Conversely, the impact of hardware EDMs is much less important in the case of the SWIFI experiments. Furthermore, the target segment (code or data) has a dramatic impact on the type of mechanisms exercised (and thus on the type of errors generated). Accordingly, these two techniques will be considered separately in the analyses carried out in Section 6.

A closer examination of the errors detected by the hardware EDMs revealed that 5.0 percent, 11.6 percent, and 1.9 percent of the errors were detected by the time-slice controller (TSC)—that is, triggering an NMI—for heavy-ion, pin-forcing, and EI, respectively. Although no NMIs were observed during the SWIFI experiments reported here, a small number of NMIs generated by the time-slice controller were observed during other experiments. No fail silence violations in the time domain (see Section 5.4) were observed during these experiments, thus demonstrating the usefulness of this error detection mechanism.

The *system software EDMs* detected the second largest amount of errors for all the ΦFI techniques. The imbalance observed in the case of heavy-ion radiation between the OS and CGRTA EDMs is amplified when using pin-forcing and EI: Almost no detections by the CGRTAs were observed for the two latter techniques. For SWIFI, on the contrary, the CGRTA EDMs dominate, but the overall impact of the system software EDMs is significantly reduced.

The *application-level (end-to-end) EDMs* detected the smallest amount of errors for all ΦFI techniques. This is opposite for the SWIFI technique. However, when these were disabled, the fail silence coverage was significantly reduced (particularly for heavy-ion radiation and SWIFI on code segment), which shows the necessity of using these mechanisms as well.

Another important outcome of the study concerned the analysis of the impact of the various EDMs on the *fail silence* property of a MARS node. The results shown for each technique in the previous tables, where three configurations involving, respectively, both (DEMC), only one (SEMC), or none (NOAM) of the end-to-end EDMs are presented, sustain the conviction that the end-to-end EDMs play a dominant role (with respect to the other EDMs) in achieving the fail silence property. To further check this view, a specific series of experiments was carried out for HI and PF for which the NMI EDMs were disabled. These experiments focused essentially on the application unit processor. The results can be summarized as follows: In both cases, almost no fail silence violations were observed for the DEMC configuration (with NMI disabled), while, as shown in Tables 5 and 6, the NOAM configuration (with NMI enabled) exhibited a significant number of fail silence violations.

6 Analysis of the Fault Injection Techniques

This section provides a detailed study of the actual impact of the fault injection techniques considered. Indeed, each set of experiments carried out using a specific fault injection technique can be considered as a kind of "benchmark" to assess the relative effectiveness of the various EDMs. On those grounds, the distribution of the sensitization among the various EDMs and failure modes, as well as among the error data observed for each experiment, constitute suitable —albeit, indirect—means to identify the similarities and differences of the error sets induced by the four injection techniques considered.

If, when applying two techniques, the observed error sets are nearly the same, then the technique that is the most expensive or the most difficult to control may be substituted by the other one. Conversely, if the error sets differ to a large extent, the fault injection techniques may instead be used to complement each other. Some insights on the additional properties characterizing the fault injection techniques considered are provided in Section 7.

7. Note that each of the two task instances executed has its own code and data segment. Therefore, injection of a single error in one of the task instances is very likely to be detected (this was also confirmed by the experiments consisting of multiple bit-flips [28]).

8. This is true for all ΦFI campaigns, except for EI campaign DEMC(3) in Table 5. As already pointed out, this campaign exhibited results which were drastically different from the other EI campaigns. Accordingly, this led us to suspect it, so we prefer to exclude these results from subsequent analyses.

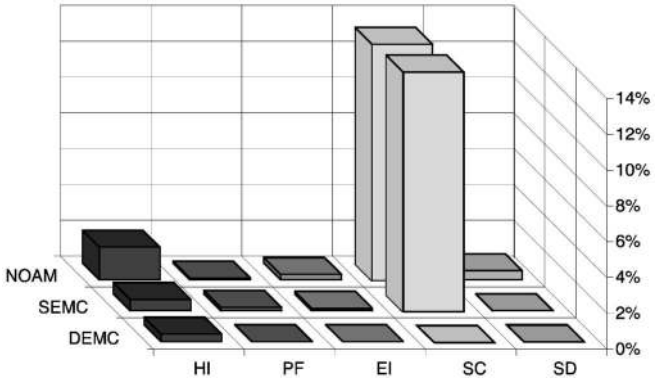

Fig. 6. Fail silence violations provoked.

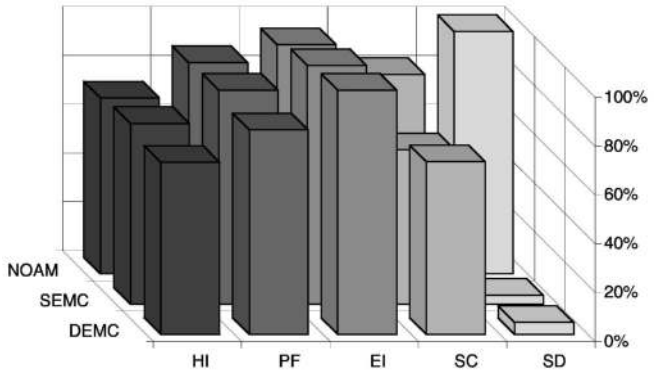

Fig. 7. Activation of the hardware EDMs.

In the sequel, we revisit the results from Section 5 by comparing the impact of the variation of the experimental configuration (combining the end-to-end EDMs) on the observation of fail silence violations, as well as of the three levels of EDMs included in the NUT—namely, hardware, system software, and end-to-end—for each injection technique. For the ΦFI techniques—heavy-ion (HI), pin forcing (PF), and electromagnetic interferences (EI)—the error reports concerning both the application unit and communication unit experiments have been merged. For SWIFI, due to the notably different impact observed, we explicitly distinguish the experiments targeting the segments of code (SC) and data (SD). Furthermore, in the sequel, we focus the analysis by considering only the actual error reports, i.e., the "No Reply" category will be disregarded and the other percentages are normalized accordingly. Also, the intermittent "IMLoss" outcomes have been transferred to the "Checksum" category (although some message losses were observed, this misbehavior was eventually caught by this error detection mechanism). More details on the results that support this analysis can be found in [48] and [33], for the ΦFI and SWIFI experiments, respectively.

6.1 Fail Silence Violations

Fig. 6 compares the impact of the fault injection techniques for the different configurations of the end-to-end EDMs on the observation of fail silence violations. As already pointed out, the impact of the SWIFI technique depends strongly on where the faults were injected in the rolling ball application software: code (SC) or data (SD) segments. Injection in the code segment generates significantly more fail silence violations. Summarizing, SC appears to be more malicious than the physical techniques both for the NOAM and SEMC configurations, but HI is more stressful for the DEMC configuration.

6.2 Hardware Error Detection Mechanisms

Fig. 7 compares the impact of the fault injection techniques on the percentages of errors that activated the hardware EDMs.

In the NOAM case, all techniques provide a large ratio of hardware error detection (more than 70 percent). Although an important percentage of hardware detections is maintained for the ΦFI techniques (and, to some extent, for SWIFI on the code segment), when the application-level EDMs are enabled, the percentages observed are significantly reduced for SD.

The significant difference observed for the two types of faults injected by the software technique is worth noting. For SC, this percentage is maintained above 60 percent, while, for SD, it is reduced to less than 5 percent. This suggests that the faults injected in the code segment provoke, rather control flow errors that (to a large extent) better simulate the consequences of hardware faults. The SD experiments generate mainly data flow errors that are different from the ones created by the other techniques. Nevertheless, it is interesting to note that SD can provide a rather high level of activation of the hardware EDMs in the NOAM configuration, i.e., when the end-to-end mechanisms are inhibited. A closer examination of the results was carried out. Table 8 summarizes some of the main differences observed.

Concerning CPU EDMs, although, out of the eight mechanisms supported, a different number of mechanisms were activated, the same three mechanisms dominated (bus error, address error, and illegal opcode) for all the fault injection techniques. The number of NMI types and of their combinations (i.e., the simultaneous occurrences of several triggering events) vary significantly for the ΦFI techniques.[9] The results indicate that PF may be more effective than the other techniques in exercising hardware EDMs located outside of the CPU chip. Moreover, the most frequent NMIs observed differ: While "unavailable memory" significantly dominates for HI (more than 60 percent), "memory parity" dominates for PF and EI (more than 50 percent). Both PF and EI also exhibited a significant proportion of NMIs triggered by the TSC (more than 15 percent). The differences observed are further exemplified by the variations in the number of different types of exceptions (including CPU-related and NMI) activated during the various experiments, out of the 255 possible exceptions. This is illustrated by Fig. 8, which shows the distribution of the exceptions observed for the three ΦFI techniques considered.

However, it is worth noting that, for EI, most of the experiments exercised CPU EDMs (especially when using the probe without antennas), which reveals the very restricted spectrum of the type of errors generated by this technique. However, the variation in the error set was

9. As shown in Table 7, no NMIs were observed for the SWIFI experiments reported here.

TABLE 8
Detailed Analysis of the Hardware EDMs Activated by the Injection Techniques

Hardware EDMs	Heavy-ion	Pin-forcing	Electrom. Interf.	SWIFI
# of CPU EDMs activated	7	4	5	5
# of NMI types (and combinations)	13 (26)	17 (34)	10 (16)	0
# of exceptions activated	73	65	14	Data not available

somewhat enhanced when the antennas were used (see Table 6).

6.3 System Software Error Detection Mechanisms

Fig. 9 gathers the activations of the system software EDMs induced by the injection techniques. First, it is worth noting that whether the end-to-end EDMs are enabled or not has no significant impact on these results. Besides, the results show some differences ranging from 0.4 percent for EI (SEMC)[10] to 7.6 percent for HI (NOAM), the system software mechanisms were the second largest EDM activated by the ΦFI techniques.

For SWIFI, here again, the results observed for SC and SD vary significantly: The percentages of activations observed for SC (ranging from 4.6 to 6.5 percent) are comparable with those corresponding to the ΦFI techniques, while the percentages observed for SD are always less than 0.1 percent. As already pointed out, another notable difference is the fact that OS occurrences dominate for all the ΦFI techniques while CGRTAs dominate for SC. A closer look at these results identified the "internal FIFO empty" as the far more frequently activated OS EDM for the ΦFI experiments (more than 23 percent for HI and 8 percent for PF and EI of OS occurrences). The assertions most often violated during the SWIFI experiments are 1) the failure of the range check for a variable (about 60 percent of CGRTA occurrences) and 2) 32-bit multiplication overflow (more than 30 percent).

6.4 End-to-End Error Detection Mechanisms

Fig. 10 depicts the results concerning the end-to-end EDMs. The figure exhibits a very important difference between the results obtained from ΦFI and SWIFI experiments. It is highly likely that this difference can be attached to the distinct fault/error models injected by the physical and software injection techniques considered.

These results show and confirm that errors induced by the ΦFI techniques are more prone to activate the hardware and system software EDMs and, thus, a very limited number of errors was perceived by the end-to-end EDMs (less than 3.6 percent for HI, 0.5 percent for EI, and 0.2 percent for PF).

Conversely, the end-to-end mechanisms are actually significant EDMs for SWIFI. Nevertheless, some interesting differences could be observed between SC and SD:[11]

- SD provides the highest percentage (more than 95 percent), while SC features less than 25 percent,
- double execution significantly increases the detection percentage for SC (from 12 percent to 24.3 percent).

The notable difference between SC and SD concerns the percentage of error detections obtained either by the *message checksum* or by the *double execution check* in the case of the DEMC configuration (see also Table 7, for details). For SC, double execution detects most errors (60 percent of end-to-end detections), while, for SD, message checksum EDM dominates (more than 70 percent). The explanation of this last observation is that the message check is done at the *start* of an application task, while the double execution check is carried out only *after* the completion of the second task instance and, thus, is likely to be dominated by message checksum.

7 Characterization of the Fault Injection Techniques

We briefly comment in this section on some additional important issues that also have to be taken into account when selecting a fault injection technique. In addition to *fault representativeness* (i.e., the plausibility of the supported fault model with respect to actual faults) that is one concern that is often raised in conjunction with fault injection experiments and for which we provided some objective insights in the previous section, a wide range of criteria can be considered to assess the merits of the fault injection techniques (e.g., see [3], [6], [9]). Without any claim of an exhaustive analysis, we have considered the following eight basic properties: *reachability*, *controllability*, with respect to *space* and *time*, *repeatability* (with respect to experiments), *reproducibility* (with respect to results), *nonintrusiveness*, the possibility for *time measurement* (e.g., error detection latency), and the *efficacy* to generate significant experiments.

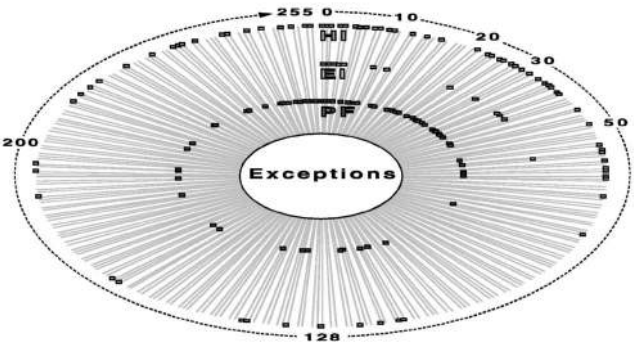

Fig. 8. Distribution of exceptions provoked by the HI, PF, and EI techniques.

10. The 0 percent value shown in Table 6 for EI in the DEMC configuration should preferably be ignored (see Section 5.5 and the comment in footnote 8).

11. Remember that the figures considered in this section for SWIFI were normalized after discarding the "No Reply" category shown in Table 7.

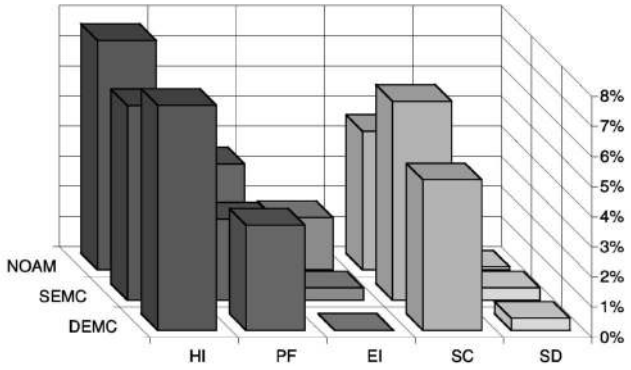

Fig. 9. Activation of the system software EDMs.

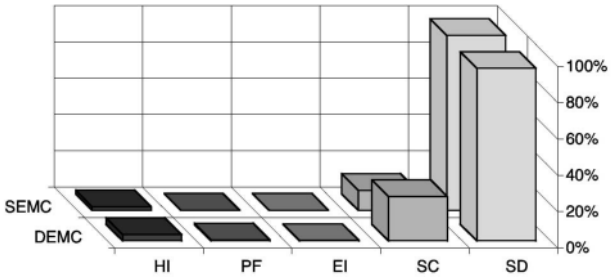

Fig. 10. Activation of the end-to-end EDMs.

A characterization of the considered fault injection techniques based on these eight basic properties is shown in Table 9 and explained hereafter. For each property, the techniques are graded on the scale none, low, medium, and high. It is worth noting that, although it is quite generic in scope, this analysis also builds upon insights gained during the experiments carried out on the MARS system. Note again that this characterization is meant to complement the analysis of the impact of the injected faults, i.e., the errors that are produced. Finally, it is worth pointing out that such a study is mainly a relative issue; accordingly, grading is very dependent of the set of techniques being assessed; indeed, different grading could be obtained should other techniques be considered.

7.1 Reachability

We consider the reachability property attached to a fault injection technique to be defined as the ability to reach possible fault locations in the ICs that implement the target system.

From that perspective, heavy-ion radiation definitely surpasses the other techniques as faults are actually injected directly at the level of the physical devices that constitute the irradiated circuit.

In the experiments conducted, pin-level fault injection was focused on the digital input/output signals. Accordingly, the corruption of data at the level of internal devices is only indirect. This is why pin-level injection has been rated with medium reachability (IC pins are targeted, but injection cannot be focused on internal devices). Nevertheless, varying reachability is obtained depending on the level of integration for the target system: It is definitely much lower for highly integrated systems.

When using antennas, EI has similar physical reachability as pin-level injection as most faults probably are injected via the digital input/output signals. However, faults may also impact internally the ICs as a result of disturbances propagated through the power supply lines.

SWIFI at preruntime directly corrupts the information (code or data) that will be stored in memory devices. Still, as opposed to runtime techniques (e.g., see [9]), this technique cannot specifically target the lower hardware layers (e.g., processor registers). Also, the injection targets are limited to information explicitly processed by the software layers of the computing system. Also, it provides low reachability with respect to peripheral ICs. This is why the technique was graded "low to medium" with respect to this property.

7.2 Controllability

In this section, we consider controllability with respect to both the space and time domains. The space domain relates to the ability to control which of the reachable fault locations are actually injected. The time domain corresponds to controlling the instant when faults are injected.

Heavy-ion radiation has low controllability for the space domain. In practice, faults could be confined to specific blocks of a circuit if the rest of the circuit is shielded. However, shielding was not used in this study. The time of the injection of a fault cannot be controlled as the decay of the Cf-252 source is governed by a random process.

Pin-level fault injection features good controllability in the space domain. Indeed, selected ICs and pins can be targeted. Basically, some extent of time domain controllability can be achieved. Nevertheless, it may be hampered by the problem of synchronizing the fault injection with the

TABLE 9
Properties of the Fault Injection Techniques Used

Properties	Heavy-ion	Pin-level	Electromagnetic Interf. (with probe)	SWIFI (pre-runtime)
Reachability	high	medium	medium	low to medium
Controllability wrt space	low	high	low	high
Controllability wrt time	none	low to medium	low	medium to high
Repeatability	none to low	medium to high	none to low	high
Reproducibility	medium to high	high	low	high
Non-intrusiveness	low	medium	high	high
Time measurement	low to medium	high	low	medium to high
Efficacy	high	high	high	low

activity of the system, especially when the clock frequency of the target system is high. Actually, controllability is very much dependent upon the level of integration and clock speed of the target system. For the MARS system, which uses a mix of VLSI, LSI, MSI, and SSI circuits and moderate clock speeds, controllability was, in fact, high. Nevertheless, high integration levels and high clock speed are the major limitations for the use of such a technique in modern digital system designs.

Electromagnetic interferences feature low controllability in the space domain because faults may be injected in circuits surrounding the target circuit. The time of injection can (to some extent) be synchronized with system activity, but it is difficult to determine exactly when a fault is injected.

SWIFI at preruntime procures a very high level of controllability as it can focus selectively on specific code and data segments. However, injection is limited to information processed by the software layers of the computing system. It is also worth pointing out that it procures less time-domain controllability than runtime techniques; accordingly, it was graded "medium to high" with respect to this property.

7.3 Repeatability

Repeatability refers to the ability to repeat experiments exactly or with a very high degree of accuracy. This property is highly desirable, particularly when the aim of the experiments is to remove potential design/implementations faults in the fault tolerance mechanisms (e.g., see [5]). Repeatability requires a high degree of controllability in both the space and the time domains.

Preruntime SWIFI achieves a very high level of repeatability. Basically, the time-triggered architecture of MARS made it possible to carry out a deterministic series of experiments.

For pin-level injection, it is possible to accurately reproduce the injection of a selected fault with MESSALINE. Still, due to limited synchronization capabilities, this does not necessarily imply that the errors being provoked are the same. Such a difficulty in reproducing an experiment is even more stringent in the case of a distributed architecture because of the problem associated with controlling and synchronizing the activities in multiple computers. However, the time-triggered architecture of the MARS system greatly facilitated such a synchronization.

Repeatability is very low, or nonexistent, for electromagnetic interferences and heavy-ion injection due to low controllability.

7.4 Reproducibility

Reproducibility refers to the ability to reproduce results statistically for a given set-up. Reproducibility of results is an absolute requirement to ensure the credibility of fault injection experiments. Repeatability normally implies reproducibility; if we can control an experiment exactly, then it is always possible to also reproduce the results. However, reproducibility can be achieved without repeatability.

The experiments conducted in this study showed that heavy-ion radiation produces results that are statistically reproducible among different specimens of the target circuits. However, the lessons learned from this study also show that statistical reproducibility is difficult to obtain for the electromagnetic interferences experiments (see Section 5.3).

7.5 Nonintrusiveness

This relates to the property of avoiding or minimizing any undesired impact of fault injection on the behavior of the target system. The heavy-ion radiation technique has low nonintrusiveness since the target IC has to be removed from the main board and inserted into a vacuum chamber. This may cause unexpected delays in the data paths of the associated logical signals.

This problem is significantly lower when using the pin-forcing technique.[12] Nevertheless, even in the case of forcing, the extra load capacitances on the targeted logical signals that are introduced by the connection of the injection probes may also alter the timing characteristics. The nonintrusiveness is therefore graded a medium for pin-forcing.

The EI technique with the probe features high nonintrusiveness since no physical connection is required between the probe and target IC, although there may be difficulties attached to actually focusing the injection to a specific IC.

As opposed to most runtime SWIFI techniques (e.g., see [8], [9], [11]), the preruntime SWIFI technique used here features minimal intrusiveness.

7.6 Time Measurement

The acquisition of timing information associated to the monitored events (e.g., measurement of error detection latency) is an important outcome from fault injection experiments (e.g., see [49]).

For heavy-ion radiation, such measurements rely on the use of the golden chip technique [2]. This requires the target IC to be operated synchronously with a reference IC. However, this may not be possible for ICs with nondeterministic external behavior (caused, for example, by a varying number of wait states cycles inserted during memory accesses). This technique could not be used in the experiments reported earlier.

For pin-level injection, the time of the injection of a fault (and activation of this fault as an error) can be explicitly known, thus, latency measurements does not pose a problem [1].

For EI, latency measurements are difficult. In principle, the golden chip technique could also be used in this case, but it may be difficult to confine the disturbances to the target IC only.

Basically, in preruntime SWIFI, the fault is being activated when the flipped (code or data) word is accessed. Accordingly, trace analysis can be used to relate this event with the error detections in order to make timing measurements such as error detection latency. However, this requires an extensive data log and, possibly, a tedious analysis. This was not considered in the experiments carried out on the MARS system.

Due to the inherent difficulties described above for all techniques (pin-level injection, excepted), no timing

12. One should note that the same form of intrusiveness would apply to pin-level injection when the insertion technique is used (in that case, the IC under test is removed from the target system and solid-state switches ensure its proper isolation from the rest of the system).

measurements were made for the experiments conducted on the MARS system.

7.7 Efficacy

The type of efficacy considered here concerns the testing power offered by the techniques, i.e., their ability to produce a limited number of nonsignificant experiments. A nonsignificant experiment occurs, for example, when a fault is injected into a hardware or software component which is not accessed or used by the workload executed during the experiment.

As shown in several previous studies, (e.g., see [1], [2], [43]), high efficacy is rather easy to achieve by ΦFI techniques. The key issue is to adjust the amplitude, duration, or intensity of the physical interferences to an appropriate level. This was also confirmed by the experiments carried out on the MARS system. Indeed, it was easy to devise an experimental protocol ensuring that all ΦFI experiments result in significant experiments leading to actual, albeit possibly empty, error reports.

Conversely, as it focuses on specific information stored in the memory, the SWIFI technique is very prone to generating nonsignificant experiments. The reason for this is that random injections into the code or data segment generate many errors that either become overwritten or remain latent throughout the experiment. As already pointed out in Section 5.4, this concern is reported in many other related studies and was observed also in the experiments conducted on MARS. Furthermore, as one could anticipate, the efficacy of the preruntime technique used here highly depends on the memory segment targeted (either code or data) and also varies according to the type of EDMs that are actually enabled.

The efficacy of SWIFI techniques could be improved by conducting preinjection analysis that determine which parts of the software (variables, constants, subroutines, etc.) are sensitive to fault injection. Examples of research in this direction can be found in [17] and [50].

8 Conclusion

This paper summarized the study concerning the analysis of the impact of four fault injection techniques that was deduced from the controlled experiments carried out on the MARS architecture. Three physical fault injection techniques—heavy-ion radiation, pin-forcing, and electromagnetic interferences—and preruntime software-implemented fault injection (SWIFI) were applied using the same hardware/software set-up.

First, it is worth pointing out that these extensive sets of experiments significantly contributed to getting confidence in the ability of the MARS nodes to implement the "fail silence" property. It was shown that, beyond the hardware and system software error detection mechanisms (EDMs), the application-level (end-o-end) EDMs are necessary for achieving a very high coverage on the fail silence assumption. Indeed, although the time-slice controller effectively prevents fail silence violations in the time domain, fail silence violations in the value domain were observed for all four injection techniques when double execution of tasks was not used.

The other major outcome from the work that is reported in this paper concerns the comparative analysis of the fault injection techniques considered. We proposed a detailed study of the impact of the fault injection techniques on the basis of the way they activated the various EDMs of the MARS architecture and also of the failure modes observed. This analysis showed that the four injection techniques are rather complementary, i.e., they generate, to a large extent, different types of errors.

In addition to this analysis, we also identified some basic properties (e.g., reachability, repeatability, controllability, etc.) that characterize the application of the fault injection techniques as a complement to support the decision to select a technique. Pin-forcing and software injection offer the highest degree of controllability and repeatability, while heavy-ion radiation features better reachability.

The results also showed that software injection using a simple bit-flip fault model is able to generate a similar error set as the physical techniques. In particular, injection in the code segment of the executed task was able to create a similar ratio of errors activating the hardware mechanisms. However, it was found that software injection in the data segment did not provoke the same type of errors as those induced by physical techniques.

As the errors provoked are seldom detected by the hardware mechanisms—in particular, no Non-Maskable Interrupts (NMIs) were reported—and by the message checksum, the errors resulting from the software injections in the code segment generate a significantly higher number of fail silence violations for the single execution configurations. Although double execution contributed to procuring a perfect coverage of the fail silence assumption for the other techniques, heavy-ion was still found stressful in the case of double execution. One likely explanation could be traced to the singular type of failure (latch-up) caused by heavy-ion. Still, this might also suggest that, although it proved to be actually virulent, the single bit-flip fault model is not fully adequate for the simulation of internal faults generated by heavy-ions and, hence, software injection could require the utilization of a more malicious fault model. More controlled experiments would be needed to get further insights.

Another issue worth commenting on is how the results of the heavy-ion method scale with the shrinking feature sizes of VLSI technology. As demonstrated in [40], approximately 99 percent of the upsets induced by heavy-ions in the 68070 CPU affect only a single bit. More recent processors are expected to be more sensitive to ionizing particles due to the reduction of the feature sizes. As indicated earlier in the paper, neutron radiation is the main cause of soft errors at ground level. Recent research (e.g., see [51]) indicates that neutrons primarily induce single bit upsets in contemporary VLSI technologies. We therefore believe that the results presented in this paper provide an interesting point of reference also for time-triggered real-time systems implemented with the most recent circuit technology (e.g., see [52]). However, more research is needed to investigate the representativeness of the Cf-252 method with respect to neutron induced soft errors for different circuit technologies.

Due to its attractive features (including high controllability and repeatability, as well as cost-effectiveness), software-implemented fault injection appears today as the preferred (and, most of the time, only) approach chosen to support a pragmatic analysis of fault-tolerant computing systems and designs. Still, as was clearly shown in our analysis, preruntime SWIFI can be ill suited to exercising error detection mechanisms implemented at the hardware

layer (e.g., none of the "NMI mechanisms" were activated by the SWIFI technique). Accordingly, from the results we have obtained that revealed the positive impact of the end-to-end mechanisms, one could be tempted to question the usefulness of including hardware error detection mechanisms at all. Such an assessment cannot be totally inferred from the results we obtained: This would necessitate running more experiments with *hardware EDMs inhibited* (besides those already conducted with NMI EDMs disabled) to be in position to assert such a statement. Another useful dimension that we could not explore in this study concerns the dynamic behavior of the system in the presence of faults. Indeed, as was shown in several studies (e.g., see [49]), fault tolerance coverage is actually a time-dependent distribution: Besides static coverage figures (the so-called coverage factor), the dependability of computer systems relies heavily on the response time (e.g., error detection latency). Besides providing valuable entries for supporting the dependability evaluation of the target fault-tolerant computer system (MARS in this case), the availability of timing measurements would have significantly enhanced the assessment of the considered fault injection techniques. In particular, it is highly likely that the errors activating the hardware error detection mechanism would exhibit much lower latencies than those being observed at the level of the software-implemented mechanisms.

ACKNOWLEDGMENTS

This work was partially supported by ESPRIT Project no. 6362: Predictably Dependable Computing Systems (PDCS-2). The authors would like to thank Professor Hermann Kopetz, the leading architect of the MARS system, for his continuing support and many valuable suggestions during this study. They also thank Dr. Jean-Claude Laprie for his inspiring comments concerning the analysis of the fault injection techniques. The anonymous reviewers are also gratefully acknowledged for the constructive comments made on an earlier version of the manuscript that greatly contributed to improving this paper.

REFERENCES

[1] J. Arlat, M. Aguera, L. Amat, Y. Crouzet, J.-C. Fabre, J.-C. Laprie, E. Martins, and D. Powell, "Fault Injection for Dependability Validation—A Methodology and Some Applications," *IEEE Trans. Software Eng.,* vol. 16, no. 2, pp. 166-182, Feb. 1990.

[2] J. Karlsson, P. Lidén, P. Dahlgren, R. Johansson, and U. Gunneflo, "Using Heavy-Ion Radiation to Validate Fault-Handling Mechanisms," *IEEE Micro,* vol. 14, no. 1, pp. 8-23, Feb. 1994.

[3] M.-C. Hsueh, T.K. Tsai, and R.K. Iyer, "Fault Injection Techniques and Tools," *Computer,* vol. 30, no. 4, pp. 75-82, Apr. 1997.

[4] J.V. Carreira, D. Costa, and J.G. Silva, "Fault Injection Spot-Checks Computer System Dependability," *IEEE Spectrum,* vol. 36, pp. 50-55, Aug. 1999.

[5] J. Arlat, J. Boué, and Y. Crouzet, "Validation-Based Development of Dependable Systems," *IEEE Micro,* vol. 19, no. 4, pp. 66-79, July/Aug. 1999.

[6] J. Arlat, "Fault Injection for the Experimental Validation of Fault-Tolerant Systems," *Proc. Workshop Fault-Tolerant Systems,* pp. 33-40, 1992.

[7] J.H. Barton, E.W. Czeck, Z.Z. Segall, and D.P. Siewiorek, "Fault Injection Experiments Using FIAT," *IEEE Trans. Computers,* vol. 39, no. 4, pp. 575-582, Apr. 1990.

[8] G.A. Kanawati, N.A. Kanawati, and J.A. Abraham, "FERRARI: A Flexible Software-Based Fault and Error Injection System," *IEEE Trans. Computers,* vol. 44, no. 2, pp. 248-260, Feb. 1995.

[9] J. Carreira, H. Madeira, and J.G. Silva, "Xception: A Technique for the Experimental Evaluation of Dependability in Modern Computers," *IEEE Trans. Software Eng.,* vol. 24, no. 2, pp. 125-136, Feb. 1998.

[10] D.T. Stott, G. Ries, M.-C. Hsueh, and R.K. Iyer, "Dependability Analysis of a High-Speed Network Using Software-Implemented Fault Injection and Simulated Fault Injection," *IEEE Trans. Computers,* vol. 47, no. 1, pp. 108-119, Jan. 1998.

[11] J. Arlat, J.-C. Fabre, M. Rodríguez, and F. Salles, "Dependability of COTS Microkernel-Based Systems," *IEEE Trans. Computers,* vol. 51, no. 2, pp. 138-163, Feb. 2002.

[12] H. Madeira, D. Costa, and M. Vieira, "On the Emulation of Software Faults by Software Fault Injection," *Proc. Int'l Conf. Dependable Systems and Networks (DSN-2000),* pp. 417-426, 2000.

[13] R. Chillarege and N.S. Bowen, "Understanding Large System Failures—A Fault Injection Experiment," *Proc. 19th Int'l Symp. Fault-Tolerant Computing (FTCS-19),* pp. 356-363, 1989.

[14] M. Daran and P. Thévenod-Fosse, "Software Error Analysis: A Real Case Study Involving Real Faults and Mutations," *Proc. Int'l Symp. Software Testing and Analysis (ISSTA '96),* pp. 158-171, 1996.

[15] A. Mukherjee and D.P. Siewiorek, "Measuring Software Dependability by Robustness Benchmarking," *IEEE Trans. Software Eng.,* vol. 23, no. 6, pp. 366-378, June 1997.

[16] P. Koopman and J. DeVale, "Comparing the Robustness of POSIX Operating Systems," *Proc. 29th Int'l Symp. Fault-Tolerant Computing (FTCS-29),* pp. 30-37, 1999.

[17] J. Güthoff and V. Sieh, "Combining Software-Implemented and Simulation-Based Fault Injection into a Single Fault Injection Method," *Proc. 25th Int'l Symp. Fault-Tolerant Computing (FTCS-25),* pp. 196-206, 1995.

[18] C.R. Yount and D.P. Siewiorek, "A Methodology for the Rapid Injection of Transient Hardware Errors," *IEEE Trans. Computers,* vol. 45, no. 8, pp. 881-891, Aug. 1996.

[19] J. Christmansson, M. Hiller, and M. Rimén, "An Experimental Comparison of Fault and Error Injection," *Proc. Ninth Int'l Symp. Software Reliability Eng., (ISSRE '98),* pp. 369-378, 1998.

[20] P. Folkesson, S. Svensson, and J. Karlsson, "A Comparison of Simulation Based and Scan Chain Implemented Fault Injection," *Proc. 28th Int'l Symp. Fault-Tolerant Computing (FTCS-28),* pp. 284-293, 1998.

[21] Z. Kalbarczyk, G. Ries, M.S. Lee, Y. Xiao, J. Patel, and R.K. Iyer, "Hierarchical Approach to Accurate Fault Modeling for System Evaluation," *Proc. Int'l Computer Performance and Dependability Symp. (IPDS '98),* pp. 249-258, 1998.

[22] C. Constantinescu, "Assessing Error Detection Coverage by Simulated Fault Injection," *Proc. Third European Dependable Computing Conf. (EDCC-3),* pp. 161-170, 1999.

[23] J.L. Aidemark, J.P. Vinter, P. Folkesson, and J. Karlsson, "GOOFI: A Generic Fault Injection Tool," *Proc. 2001 Int'l Conf. Dependable Systems and Networks (DSN-2001),* pp. 83-88, 2001.

[24] J. Reisinger, A. Steininger, and G. Leber, "The PDCS Implementation of MARS Hardware and Software," *Predictably Dependable Computing Systems,* B. Randell, J.-C. Laprie, H. Kopetz and B. Littlewood, eds., pp. 209-224, Berlin: Springer, 1995.

[25] H. Kopetz and G. Bauer, "The Time-Triggered Architecture," *Proc. IEEE,* vol. 91, no. 1, pp. 112-126, Jan. 2003.

[26] D. Powell, "Failure Mode Assumptions and Assumption Coverage," *Proc. 22nd Int'l Symp. Fault-Tolerant Computing (FTCS-22),* pp. 386-395, 1992.

[27] J. Karlsson, P. Folkesson, J. Arlat, Y. Crouzet, G. Leber, and J. Reisinger, "Application of Three Physical Fault Injection Techniques to the Experimental Assessment of the MARS Architecture," *Dependable Computing for Critical Applications (Proc. Fifth IFIP Working Conf. Dependable Computing for Critical Applications: DCCA-5),* R.K. Iyer, M. Morganti, W.K. Fuchs and V. Gligor, eds., pp. 267-287, 1998.

[28] E. Fuchs, "An Evaluation of the Error Detection Mechanisms in MARS Using Software-Implemented Fault Injection," *Proc. Second European Dependable Computing Conf. (EDCC-2),* pp. 73-90, 1996.

[29] E. Fuchs, "Validating the Fail-Silence of the MARS Architecture," *Dependable Computing for Critical Applications (Proc. Sixth IFIP Int'l Working Confer. Dependable Computing for Critical Applications: DCCA-6),* M. Dal Cin, C. Meadows and W.H. Sanders, eds., pp. 225-247, 1998.

[30] D. Powell, "Distributed Fault-Tolerance—Lessons from Delta-4," *IEEE Micro,* vol. 14, no. 1, pp. 36-47, Feb. 1994.

[31] Philips Semiconductors, *SCC68070 User Manual 1991, Part 1—Hardware,* 1992.
[32] A. Vrchoticky, "Modula/R Language Definition," Technical Report no. 2/92, Institut für Technische Informatik, Technische Universität Wien, 1992.
[33] H. Kopetz, P. Holzer, G. Leber, and M. Schindler, "The Rolling Ball on MARS," Research Report no. 13/91, Vienna Univ. of Technology, 1991.
[34] S. Poledna, A. Burns, A. Wellings, and P. Barrett, "Replica Determinism and Flexible Scheduling in Hard Real-Time Dependable Systems," *IEEE Trans. Computers,* vol. 49, no. 2, pp. 100-111, Feb. 2000.
[35] H. Kopetz, G. Grünsteidl, and J. Reisinger, "Fault-Tolerant Membership in a Synchronous Distributed Real-Time System," *Dependable Computing for Critical Applications,* A. Avizienis and J.-C. Laprie, eds., pp. 411-429, Vienna: Springer-Verlag, 1991.
[36] C. Constantinescu, "Impact of Deep Submicron Technology on Dependability of VLSI Circuits," *Proc. Int'l Conf. Dependable Systems and Networks (DSN-2002),* pp. 205-209, 2002.
[37] P. Shivakumar, M. Kistler, S.W. Keckler, D. Burger, and L. Alvisi, "Modeling the Effect of Technology Trends on the Soft Error Rate of Combinatorial Logic," *Proc. Int'l Conf. Dependable Systems and Networks (DSN-2002),* pp. 389-398, 2002.
[38] E. Normand, "Single Event Upset at Ground Level," *IEEE Trans. Nuclear Science,* vol. 43, no. 6, pp. 2742-2750, Feb. 1996.
[39] P. Hazucha, "Background Radiation and Soft Errors in CMOS Circuits," doctoral dissertation, no. 638, Linköping Univ., Sweden, 2000.
[40] R. Johansson, "On Single Event Upset Error Manifestation," *Proc. First European Dependable Computing Conf. (EDCC-1),* pp. 217-231, 1994.
[41] C.J. Walter, "Evaluation and Design of an Ultra-Reliable Distributed Architecture for Fault Tolerance," *IEEE Trans. Reliability,* vol. 39, no. 4, pp. 492-499, Oct. 1990.
[42] H. Madeira, M. Rela, F. Moreira, and J.G. Silva, "RIFLE: A General Purpose Pin-Level Fault Injector," *Proc. First European Dependable Computing Conf. (EDCC-1),* pp. 199-216, 1994.
[43] R.J. Martínez, P.J. Gil, G. Martín, C. Pérez, and J.J. Serrano, "Experimental Validation of High-Speed Fault-Tolerant Systems Using Physical Fault Injection," *Dependable Computing for Critical Applications (Proc. Seventh IFIP Working Conf. Dependable Computing for Critical Applications: DCCA-7),* C.B. Weinstock and J. Rushby, eds., pp. 249-265, Jan. 1999.
[44] Y. Crouzet, P. Thévenod-Fosse, and H. Waeselynck, "Validation of Software Testing by Fault Injection: The SESAME Tool," *Proc. 11th Conf. Reliability and Maintainability,* pp. 551-559, 1998.
[45] J.M. Voas and G. McGraw, *Software Fault Injection.* New York: Wiley Computer Publishing, 1998.
[46] E. Fuchs, "Sofware Implemented Fault Injection," PhD dissertation, Vienna Univ. of Technology, Austria, 1996.
[47] W.-L. Kao, R.K. Iyer, and D. Tang, "FINE: A Fault Injection and Monitoring Environment for Tracing the UNIX System Behavior under Faults," *IEEE Trans. Software Eng.,* vol. 19, no. 11, pp. 1105-1118, Nov. 1993.
[48] P. Folkesson, "Experimental Validation of a Fault-Tolerant System Using Physical Fault Injection," Licenciate of Eng. thesis, Chalmers Univ. of Technology, Göteborg, Sweden, 1996.
[49] J. Arlat, A. Costes, Y. Crouzet, J.-C. Laprie, and D. Powell, "Fault Injection and Dependability Evaluation of Fault-Tolerant Systems," *IEEE Trans. Computers,* vol. 42, no. 8, pp. 913-923, Aug. 1993.
[50] A. Benso, M. Rebaudengo, I. Impagliazzo, and P. Marmo, "Fault-List Collapsing for Fault Injection Experiments," *Proc. Ann. Reliability & Maintainability Symp. (RAMS '98),* pp. 383-388, 1998.
[51] S. Satoh, Y. Tosaka, and S.A. Wender, "Geometric Effect of Multiple-Bit Soft Errors Induced by Cosmic Ray Neutrons on DRAM's," *IEEE Electron Device Letters,* vol. 21, no. 6, pp. 310-312, 2000.
[52] H. Kopetz, "Time-Triggered Real-Time Computing," *Proc. IFAC World Congress,* 2002, http://manuals.elo.utfsm.cl/conferences/15-IFAC/data/content/05006/5006.pdf.

Jean Arlat (M'80) received the Engineer degree from the National Institute of Applied Sciences of Toulouse in 1976, and the PhD and *Docteur ès-Sciences* degrees from the National Polytechnic Institute of Toulouse in 1979 and 1990, respectively. He is *Directeur de Recherche* of CNRS, the French National Organization of Scientific Research and currently leads the research group on Dependable Computing and Fault Tolerance at LAAS-CNRS. His research interests focus on the dependability of hardware-and-software fault-tolerant systems and of software executives, including both analytical modeling and experimental approaches. He chairs the IFIP Working Group 10.4 on Dependable Computing and Fault Tolerance. He is a member of the ACM, IEEE, and SEE Working Group on Dependable Computing.

Yves Crouzet received the Engineer degree from the Higher National School of Electronics, Electrical Engineering, Computer Science, and Hydraulics, Toulouse, in 1975 and the *Docteur-Ingénieur* degree from the National Polytechnic Institute, Toulouse, in 1978. He is currently *Chargé de Recherche* of CNRS in the Dependable Computing and Fault Tolerance group. Since 1982, his main research interests have concerned the experimental validation of dependable systems by fault injection and the experimental validation of software testing methods by mutation analysis.

Johan Karlsson received the MS degree in electrical engineering in 1982 and the PhD degree in computer engineering in 1990, both from Chalmers University of Technology. He is an associate professor in the Department of Computer Engineering at Chalmers. His current research interests include low-cost fault tolerance techniques for embedded systems, robust real-time kernels, and the use of fault injection for validation of fault tolerance. He is a member of the IEEE and the IEEE Computer Society.

Peter Folkesson received the MS degree in computer science and engineering in 1993 and the PhD degree in computer engineering in 1999, both from Chalmers University of Technology. He is currently an assistant professor in the Department of Computer Engineering at Chalmers. His research activities involve assessing and developing techniques for experimental dependability validation of computer systems and investigating cost-effective techniques for improving the dependability of computer systems. He is a member of the IEEE.

Emmerich Fuchs received the MSc and PhD degrees in computer science from the Vienna University of Technology, where his research focused on worst-case execution time analysis and software-implemented fault injection for distributed real-time systems. He was one of the three founding members of DECOMSYS-Dependable Computer Systems GmbH, a Vienna University of Technology spin-off which is one of the key development members of the FlexRay Consortium. Besides his role as a managing partner in DECOMSYS, he currently works as an administrator for the FlexRay Consortium. He is a member of the IEEE Computer Society.

Günther H. Leber received the Dipl-Ing (MSc) degree in computer science in 1992 from the Technical University of Vienna. There he worked in the Department of Real-Time Systems on physical fault-injection experiments. Since August 1997, he has been affiliated with Adcon Telemetry AG—a supplier of wireless telemetry and data transmission technology—where he works as an embedded systems software engineer. He is a member of the ACM, IEEE, and IEEE Computer Society.