A Survey on Fault Injection Techniques

Haissam Ziade [1], Rafic Ayoubi [2], and Raoul Velazco [3]
[1] Faculty of Engineering I, Lebanese University, Lebanon
[2] Faculty of Engineering, University of Balamand, Lebanon
[3] IMAG Institute, TIMA Laboratory, France

Abstract: *Fault tolerant circuits are currently required in several major application sectors. Besides and in complement to other possible approaches such as proving or analytical modeling whose applicability and accuracy are significantly restricted in the case of complex fault tolerant systems, fault-injection has been recognized to be particularly attractive and valuable. Fault injection provides a method of assessing the dependability of a system under test. It involves inserting faults into a system and monitoring the system to determine its behavior in response to a fault. Several fault injection techniques have been proposed and practically experimented. They can be grouped into hardware-based fault injection, software-based fault injection, simulation-based fault injection, emulation-based fault injection and hybrid fault injection. This paper presents a survey on fault injection techniques with comparison of the different injection techniques and an overview on the different tools.*

Keywords: *Fault tolerance, fault injection, fault simulation, VLSI circuits, fault injector, VHDL fault models.*

Received May 19, 2003; accepted October 13, 2003

1. Introduction

A system may not always perform the function it is intended for. The causes and consequences of deviations from the expected function of a system are called the *factors* to dependability:

- *Fault* is a physical defect, imperfection, or flaw that occurs within some hardware or software component.
- *Error* is a deviation from accuracy or correctness and is the manifestation of a fault.
- *Failure* is the non-performance of some action that is due or expected.

When a fault causes an incorrect change in a machine stage, an error occurs. Although a fault remains localized in the affected code or circuitry, multiple errors can originated from one fault site and propagate throughout the system. When the fault-tolerance mechanisms detect an error, they may initiate several actions to handle the faults and contain its errors. Otherwise, the system eventually malfunctions and a failure occurs.

Due to the evolutions of the technologies, the probability of faults occurring in integrated circuits is noticeably increasing. The interest for integrated on-line fault detection mechanisms and/or fault tolerance is therefore rapidly increasing for circuits designed in deep sub-micron technologies. The selection of the right mechanisms to integrate in a circuit requires the definition of the type of faults prone to occur and a detailed knowledge of their potential impact on the circuit behavior. Designing a circuit with a set of carefully selected fault detection mechanisms is not sufficient to ensure that all critical effects of faults are avoided, especially because the implemented mechanisms are in general not able to catch all possible faults. Trade-offs have also to be made during the design phase between the fault coverage obtained for different types of faults and the various induced costs (in terms of area or clock frequency, but also for example in terms of design time and potential impacts on the application execution if techniques based on time redundancy is used). Making such choices, it is necessary to ensure that the faults which are eventually not detected or tolerated by the implemented mechanisms do not have critical effects. However, the level of criticality of the effect can often only be evaluated in the application environment, taking into account the real-time interactions of the circuit with the other system elements. The analysis of the fault effects must therefore often take into account not only the internal circuit description but also the external system definition. The final assessment of the circuit and system dependability is classically done using *fault injections* on a system prototype. In terms of cost and time, it becomes therefore crucial to perform a thorough analysis of the failure modes of the circuit if possible early in the design process and at least before any manufacturing [17, 31].

1.1. Fault Category

A fault as a deviation in a hardware or software component from its intended function can arise during all stages in a computer system design process:

Specification, design, development, manufacturing, assembly, and installation throughout its operational life. Most faults that occur before full system deployment are discovered and eliminated through testing. Faults that are not removed can reduce a system's dependability when it is embedded into the system.

Hardware/Physical Fault that arise during system operation are best classified by their duration: Permanent, transient, or intermittent.

- *Permanent faults:* Caused by irreversible component damage, such as a semiconductor junction that has shorted out because of thermal aging, improper manufacture, or misuse. Since it is possible that a chip in a network card that burns causing the card to stop working, recovery can only be accomplished by replacing or repairing the damaged component or subsystem.
- *Transient faults:* Triggered by environmental conditions such as power-line fluctuation, electromagnetic interference, or radiation. These faults rarely do any lasting damage to the component affected, although they can induce an erroneous state in the system. According to several studies, transient faults occur far more often than permanent ones, and are also far harder to detect.
- *Intermittent faults:* Caused by unstable hardware or varying hardware states. They can be repaired by replacement or redesign.

Hardware faults of almost all types are easily injected by the devices available for the task. Dedicated hardware tools are available to flip bits on the instant at the pins of a chip, vary the power supply, or even bomb the system/chips with heavy ions-methods believed to cause faults close to real transient hardware faults. An increasingly popular software tool is a *software-implemented fault injector*, which changes bits in processor registers or memory, in this way producing the same effects as transient hardware faults. All these techniques require that a system, or at least a prototype, actually be built in order to perform the fault testing.

Software faults are always the consequence of incorrect design, at specification or at coding time. Every software engineer knows that a software product is bug free only until the next bug is found. Many of these faults are latent in the code and show up only during operation, especially under heavy or unusual workloads and timing contexts.

Since they are a result of bad design, it might be supposed that all software faults would be permanent. Interestingly, practice shows that despite their permanent nature, their behavior is transient; that is, when a bad behavior of the system occurs, it cannot be observed again, even if great care is taken to repeat the situation in which it occurred. Such behavior is commonly called a failure of the system. The subtleties of the system state may mask the fault, as when the bug is triggered by very particular timing relationships between several system components, or by some other rare and irreproducible situation.

Curiously, most computer failures are blamed on either software faults or permanent hardware faults, to the exclusion of the transient and intermittent hardware types. Yet many studies show these types are much more frequent than permanent faults. The problem is that they are much harder to track down.

During the process of software development, faults can be created in every step: Requirement definition, requirement specifications, design, implementation, testing, and deployment. And these faults can be cataloged to:

- *Function faults*: Incorrect or missing implementation that requires a design change to be corrected.
- *Algorithm faults*: Incorrect or missing implementation that can be fixed without the need of design change.
- *Timing/serialization faults*: Missing or incorrect serialization of shared resources.
- *Checking fault*: Missing or incorrect validation of data, or incorrect loop, or incorrect conditional statement.
- *Assignment fault*: Values assigned incorrectly or not assigned.

2. An Overview of Fault Injection

Fault Injection is defined by Arlat [3] as the validation technique of the dependability of fault tolerant systems which consists in the accomplishment of controlled experiments where the observation of the system's behavior in presence of faults is induced explicitly by the writing introduction (injection) of faults in the system.

The fault injection techniques have been recognized for a long time as necessary to validate the dependability of a system by analyzing the behavior of the devices when a fault occurs. Several efforts have been made to develop techniques for injecting faults into a system prototype or model. Most of the developed techniques fall into five main categories:

- Hardware-based fault injection: It is accomplished at physical level, disturbing the hardware with parameters of the environment (heavy ion radiation, electromagnetic interferences, etc.), injecting voltage sags on the power rails of the hardware (power supply disturbances), laser fault injection or modifying the value of the pins of the circuit.
- Software-based fault injection (software implemented fault injection): The objective of this technique consists of reproducing at software level the errors that would have been produced upon occurring faults in the hardware.

- Simulation-based fault injection: Consists in injecting the faults in high-level models (most often, VHDL models). It allows early evaluating the system dependability when only a model of the system is available. Then it addresses different abstraction levels by using distinct description languages. A coherent environment should be provided to favor interoperability between the successive abstraction levels and to integrate the validation in the design process.
- Emulation-based fault injection: This technique has been presented as an alternative solution for reducing the time spent during simulation-based fault injection campaigns. It is based on the exploration of the use of Field Programmable Gate Arrays (FPGAs) for speeding-up fault simulation and exploits FPGAs for effective circuit emulation. This technique can allow the designer to study the actual behavior of the circuit in the application environment, taking into account real-time interactions. However, when an emulator is used, the initial VHDL description must be synthesizable.
- Hybrid fault injection: This approach mix software-implemented fault injection and hardware monitoring.

From another point of view, the fault injection techniques can be grouped into invasive and non-invasive techniques. The problem with sufficiently complex systems, particularly time dependant ones, is that it may be impossible to remove the footprint of the testing mechanism from the behavior of the system, independent of the fault injected. Invasive techniques are those which leave behind such a footprint during testing. Non-invasive techniques are able to mask their presence so as to have no effect on the system other than the faults they inject.

2.1. Fault Injection Environment

A fault injection environment typically consists of the following components:

- *Fault injector:* Injects fault into the target system as it executes commands from the workload generator.
- *Fault library: Stores* different fault types, fault locations, fault times, and appropriate hardware semantics or software structures.
- *Workload generator:* Generates the workload for the target system as input.
- *Workload library:* Stores sample workloads for the target system.
- *Controller:* Controls the experiment.
- *Monitor:* Tracks the execution of the commands and initiate data collection whenever necessary.
- *Data collector:* Performs online data collection.
- *Data analyzer:* Performs data processing and analysis.

2.2. Objectives of Fault Injection

Fault injection tries to determine whether the response of the system matches with its specifications, in presence of a defined range of faults. Normally, faults are injected in perfectly chosen system states and points, previously determined by an initial system analysis. Tester knows the design in depth and so it designs the test cases (type of faults, test points, injection time and state, etc.) based on a structural criteria and usually in a deterministic way.

Fault injection techniques provide a way for *fault removal* (the correction of potential fault tolerance deficiencies in the system) and *fault forecasting* (the evaluation of the coverage distribution – coverage factor and latency – provided by the tested system). Regarding the fault removal objective, the test should be directed to achieve a high coverage of the possible configurations of the system to be validated. In this case, the selection of the faults/errors to apply and errors to propagate is primarily based on the analysis of the model describing the system and the information flow in the simulation of the system. Regarding the fault forecasting objective, the main alternatives are either to rely on statistical testing simulating a priori the relative distribution of the classes of faults/errors or to statistically process a posteriori the results of the test sequence. The data used to carry out the statistical processing may result from available file data on the distributions and/or from results of simulation experiments.

Using these two methods, fault injection techniques can yield seven benefits:

- An understanding of the effects of real faults and thus of the related behavior of the target system in terms of functionality and performance.
- An assessment of the efficacy of the fault tolerance mechanisms included into the target system and thus a feedback for their enhancement and correction (e.g., for removing designs faults in the fault tolerance mechanisms).
- A forecasting of the faulty behavior of the target system, in particular encompassing a measurement of the efficiency (coverage) provided by the fault tolerance mechanisms.
- Estimating the failure coverage and latency (i. e timing) of fault tolerant mechanisms.
- Exploring the effects of different workloads (different input profiles and environments) on the effectiveness of fault tolerant mechanisms.
- Identifying weak links in the design: For example parts of the system within which a single fault could lead to severe consequences.
- Studying the system's behavior in the presence of faults, for example propagation of fault effects between system components or the degree of fault

isolation and determining the coverage of a given set of tests.

In practice, frequently fault removal and fault forecasting are not used separately, but one follows the other. For instance, after rejecting a system by fault forecasting testing, several fault removal tests should be applied. These new tests provide actions that will help the designer to improve the system. Then, it will be applied to another fault forecasting test, and so on.

3. Hardware-Based Fault Injection

Hardware-based fault injection involves augmenting the system under analysis with specially designed test hardware to allow for the injection of faults into the system and examine the effects. It uses additional hardware to introduce faults into the target system's hardware. Depending on the faults and their locations, hardware-implemented fault injection methods fall into two categories:

- *Hardware fault injection with contact:* The injector has direct physical contact with the target system, producing voltage or current changes externally to the target chip. Examples are methods that use pin-level active probes and socket insertion. The *probe method* is usually limited to stuck-at faults, although it is possible to attain bridging faults by placing a probe across two or more pins. *Socket insertion technique* inserts a socket between the target hardware and its circuit board. The inserted socket injects stuck-at, open, or more complex logic faults into the target hardware by forcing the analog signals that represent desired logic values onto the pins of the target hardware. The pin signals can be inverted, ANDed, or ORed with adjacent pin signals or even with previous signals on the same pin.
- *Hardware fault injection without contact:* The injector has no direct physical contact with the target system. Instead, an external source produces some physical phenomenon, such as heavy ion radiation and electromagnetic interference, causing spurious currents inside the target chip.

Hardware simulations typically occur in a high level description of the circuit. This high level description is turned into a transistor level description of the circuit, and faults are injected into the circuit. Software simulation is most often used to detect the response to manufacturing defects. The system is then simulated to evaluate the response of the circuit to that particular fault. Since this is a simulation, a new fault can then be easily injected, and the simulation is rerun to measure the response to the new fault. This consumes time to construct the model, insert the faults, and then simulate the circuit, but modifications in the circuit are easier to make than later in the design cycle. This sort of testing would be used to check a circuit early in the design cycle. These simulations are non-intrusive, since the simulation functions normally other than the introduction of the fault.

Hardware fault injections occur in actual examples of the circuit after fabrication. The circuit is subjected to some sort of interference to produce the fault, and the resulting behavior is examined. So far, this has been done with transient faults, as the difficulty and expense of introducing stuck-at and bridging faults in the circuit has not been overcome. The circuit is attached to a testing equipment which operates it and examines the behavior after the fault is injected. This consumes time to prepare the circuit and test it, but such tests generally proceed faster than simulation does. It is, rather obviously, used to test circuit just before or in production. These simulations are non-intrusive, since they do not alter the behavior of the circuit other than to introduce the fault. Special circuitry should be included to cause or simulate faults in the finished circuit; these would most likely affect the timing or other characteristics of the circuit, and therefore be intrusive.

Suppositions:

- The fault injector should have no interference with the exercised system.
- Faults should be injected at internal locations to the ICs in the exercised system.
- Faults that are injected into the system are representative of the actual faults that occur within the system. It means that both random generated and non-random generated faults can be injected into the system, and both permanent and transient faults can be injected into the system.

Benefits:

- Hardware fault injection technique can access locations that is hard to be accessed by other means. For example, the Heavy-ion radiation method can inject fault into VLSI circuits at locations which are impossible to reach by other methods.
- This technique works well for the system which needs high time-resolution for hardware triggering and monitoring.
- Experimental evaluation by injection into actual hardware is in many cases the only practical way to estimate coverage and latency accurately.
- This technique injects faults which have low perturbation.
- This technique is better suited for the low-level fault models.
- Not intrusive: No modification of the target system is required to inject faults.
- Experiments are fast.
- Experiments can be run in near real-time, allowing for the possibility of running a large number of fault injection experiments.

- Running the fault injection experiments on the real hardware that is executing the real software has the advantage of including any design faults that might be present in the actual hardware and software design.
- Fault injection experiments are performed using the same software that will run in the field.
- No model development or validation required.
- Ability to model permanent faults at the pin level.

Drawbacks:

- Hardware fault injection can introduce high risk of damage for the injected system.
- High level of device integration, multiple-chip hybrid circuit, and dense packaging technologies limit accessibility to injection.
- Some hardware fault injection methods, such as state mutation, require stopping and restarting the processor to inject a fault, it is not always effective for measuring latencies in the physical systems.
- Low portability and observability.
- Limited set of injection points and limited set of injectable faults.
- A recent paper indicates that the setup time for each experiment might, in fact, offset the time gained by the ability to perform the experiments in near real-time.
- Requires special-purpose hardware in order to perform the fault injection experiments. This hardware is used to inject faults into the processor by applying the rail voltages (representing logic one and zero) to the Input/Output (I/O) pins of the processor. Also, if the processor contains appropriate special-purpose hardware known as scan chains, then the external hardware could also be used to inject stuck-at-1 and stuck-at-0 faults into the internal registers of the processor. In general, this hardware can be very difficult and costly to build.
- Limited observability and controllability. At best, one would be able to corrupt the I/O pins of the processor and the internal processor registers.

Tools:

- RIFLE: A pin-level fault injection system for dependability validation developed at University of Coimbra, Portugal [22]. This system can be adapted to a wide range of target systems and the faults are mainly injected in the processor pins. The injection of the faults is deterministic and can be reproduced if needed. Faults of different nature can be injected and the fault injector is able to detect whether the injected fault has produced an error or not without the requirement of feedback circuits. RIFLE can also detect specific circumstances in which the injected faults do not affect the target system. Sets of faults with specific impact on the target system can be generated. Fault injection results showing the coverage and latency achieved with a set of simple behavior based error detection mechanisms are presented in [22]. It is shown that up to 72,5% of the errors can be detected with fairly simple mechanisms. Furthermore, for over 90% of the faults the target system has behaved according to the fail-silent model, which suggests that a traditional computer equipped with simple error detection mechanisms is relatively close to a fail-silent computer.
- FOCUS: A design automation environment developed at University of Illinois at Urbana-Champaign [9] used for analyzing a microprocessor-based jet-engine controller used in the Boeing 747 and 757 aircrafts. FOCUS uses a hierarchical simulation environment based on SPLICE for tracing the impact of transient faults. The fault from the simulation is automatically fed into the analysis-software in order to quantify the fault tolerance of the system under test. In the controller, fault detection and reconfiguration are performed by transactions over the communication link. The simulation consists of the instructions specifically designed to exercise this cross-channel communication. The level of effectiveness of the dual configuration of the system to single and multiple transient faults is measured. The results are used to identity critical design aspects from fault tolerant viewpoint. The usefulness of state transition models which describe the error propagation within the chip, enabling identification of critical fault propagation paths and the modules most sensitive to fault propagation, are shown using the tool.
- MESSALINE: A pin-level fault forcing system developed at LAAS-CNRS [3]. MESSALINE uses both active probes and sockets to conduct pin-level fault injection. It can inject stuck-at, open, bridging, and complex logical faults, among others. It can also control the length of fault existence and the frequency. It is made up of four modules: Injection module, activation module, collection module, and management module. The injection module enables injection on up to 32 injection points by means of injecting elements that support two different fault injection techniques: Forcing and insertion. The activation module ensures the proper initialization of the target system according to the elements of the A set. The readout collection module is used to collect the elements of R set. The management module is responsible for the automatic and parametrable generation of test sequence, for the run time control of its execution and for result archiving for post-test analysis.
- FIST (Fault Injection System for Study of Transient Fault Effect): Developed at the Chalmers University of Technology in Sweden [14], employs both

contact and contactless methods to create transient faults inside the target system. This tool uses heavy-ion radiation to create transient faults at random locations inside a chip when the chip is exposed to the radiation and can thus cause single- or multiple-bit-flips. FIST can inject faults directly inside a chip, which cannot be done with pin-level injections. It can produce transient faults at random locations evenly in a chip, which leads to a large variation in the errors seen on the output pins. In addition to radiation, FIST allows for the injection of power disturbance faults.

- MARS (Maintainable Real-time System): Developed at Technical University of Vienna Austria [13]. MARS system is a time-triggered, fault-tolerant, distributed system. It consists of several computer nodes communicating by means of a synchronous time division multiple access strategy. The nodes contain extra hardware and software for fault tolerance and can be configured to operate in redundancy, i.e. when two nodes execute the same task. The fundamental fault tolerance property of each processing node in the MARS system is to be fail-silent. The implementation of the fail silence property relies on numerous Error Detection Mechanisms (EDMs) at three levels: The hardware software, the system software, and the application software level.

4. Software-Based Fault Injection

Software faults are probably the major cause of system outages. Fault injection method is a possible way to assess the consequences of hidden bugs. Traditionally, software-based fault injection involves the modification of the software executing on the system under analysis in order to provide the capability to modify the system state according to the programmer's modeling view of the system. This is generally used on code that has communicative or cooperative functions so that there is enough interaction to make fault injection useful. All sorts of faults may be injected, from register and memory faults, to dropped or replicated network packets, to erroneous error conditions and flags. These faults may be injected into simulations of complex systems where the interactions are understood though not the details of implementation, or they may be injected into operating systems to examine the effects.

Software fault injections are more oriented towards implementation details, and can address program state as well as communication and interactions. Faults are mis-timings, missing messages, replays, corrupted memory or registers, faulty disk reads, and almost any other state the hardware provides access to. The system is then run with the fault to examine its behavior. These simulations tend to take longer because they encapsulate all of the operation and detail of the system, but they will more accurately capture the timing aspects of the system. This testing is performed to verify the system's reaction to introduced faults and catalog the faults successfully dealt with. This is done later in the design cycle to show performance for a final or near-final design. These simulations can be non-intrusive, especially if timing is not a concern, but if timing is at all involved the time required for the injection mechanism to inject the faults can disrupt the activity of the system, and cause timing results that are not representative of the system without the fault injection mechanism deployed. This occurs because the injection mechanism runs on the same system as the software being tested.

Suppositions:

- Faults that are injected into the system are representative of the actual faults that occur within the system.
- The additional software required to inject the faults does not affect the functional behavior of the system in response to the injected fault. Essentially, the assumption states that the software that is used to inject the fault is independent of the rest of the system, and that any faults present in the fault injection software will not affect the system under analysis.

Benefits:

- This technique can be targeted to applications and operating systems, which is difficult to be done using hardware fault injection.
- Experiments can be run in near real-time, allowing for the possibility of running a large number of fault injection experiments.
- Running the fault injection experiments on the real hardware that is executing the real software has the advantage of including any design faults that might be present in the actual hardware and software design.
- Does not require any special-purpose hardware; low complexity, low development and low implementation cost.
- No model development or validation required.
- Can be expanded for new classes of faults.

Drawbacks:

- Limited set of injection instants: At assembly instruction level, only.
- It cannot inject faults into locations that are inaccessible to software.
- Does require a modification of the source code to support the fault injection, which means that the code that is executing during the fault experiment is not the same code that will run in the field.

- Limited observability and controllability. At best, one would be able to corrupt the internal processor registers (as well as locations within the memory map) that are visible to the programmer, traditionally referred to as the programmer's model of the processor. So faults cannot be injected in the processor pipeline or instruction queue for example.
- Very difficult to model permanent faults.
- Related to four, execution of the fault injection software could affect the scheduling of the system tasks in such a way as to cause hard, real-time deadlines to be missed, which violates assumption two.

We can categorize software injection methods on the basis of when the faults are injected: During *compile-time* or during *run-time*.

To inject faults at compile-time, the program instruction must be modified before the program image is loaded and executed. Rather than injecting faults into the hardware of the target system, this method injects errors into the source code or assembly code of the target program to emulate the effect of hardware, software, and transient faults. The modified code alters the target program instructions, causing injection. Injection generates an erroneous software image, and when the system executes the fault image, it activates the fault.

This method requires the modification of the program that will evaluate fault effect, and it requires no additional software during runtime. In addition, it causes no perturbation to the target system during execution. Because the fault effect is hard-coded, engineers can use it to emulate permanent faults. This method's implementation is very simple, but it does not allow the injection of faults as the workload program runs.

During run-time, a mechanism is needed to trigger fault injection. Commonly used triggering mechanisms include:

- Time-out: In this simplest of techniques, a timer expires at a predetermined time, triggering injection. Specifically, the time-out event generates an interrupt to invoke fault injection. The timer can be a hardware or software timer.
- Exception/trap: In this case, a hardware exception or a software trap transfer control to the fault injector. Unlike time-out, exception/trap can inject the fault whenever certain events or conditions occur. For example, a software trap instruction inserted into a target program will invoke the fault injection before the program executes a particular instruction. A hardware exception invokes injection when a hardware observed event occurs (when a particular memory location is accessed, for example). Both mechanisms must be linked to the interrupt handler vector.
- Code insertion: In this technique, instructions are added to the target program that allows fault injection to occur before particular instructions, much like the code-modification method. Unlike code modification, code insertion performs fault injection during runtime and adds instructions rather than changing original instructions. Unlike the trap method, the fault injector may exist as part of the target program and run at user mode rather than system mode.

Tools:

- FERRARI (Fault and Error Automatic Real-Time Injection): Developed at the University of Texas at Austin [19], uses software traps to inject CPU, memory, and bus faults. Ferrari consists of four components: The initializer and activator, the user information, the fault-and-error injector, and the data collector and analyzer. The fault-and-error injector uses software trap and trap handling routines. Software traps are triggered either by the program counter when it points to the desired program locations or by a timer. When the traps are triggered, the trap handling routines inject faults at the specific fault locations, typically by changing the content of selected registers or memory locations to emulate actual data corruptions. The faults injected can be those permanent or transient faults that result in an address line error, a data line error, and a condition bit error.
- FTAPE (Fault Tolerance and Performance Evaluator): Developed at the University of Illinois [30]. Engineers can inject faults into user-accessible registers in CPU modules, memory locations, and the disk subsystem. The faults are injected as bit-flips to emulate error as a result of faults. Disk system faults are injected by executing a routine in the driver code that emulates I/O errors (bus error and timer error, for example). Fault injection drivers added to the operating system inject the faults, so no additional hardware or modification of application code is needed. A synthetic workload generator creates a workload containing specified amounts of CPU, memory, and I/O activity, and faults are injected with a strategy that considers the characteristics of the workload at the time of injection (which components are experiencing the greatest amount of workload activity, for example).
- FIAT (Fault Injection-based Automated Testing): Environment developed at Carnegie Mellon University [26]. FIAT is an automated real-time distributed accelerated fault injection environment. The FIAT environment provides experimenters with facilities for defining fault classes (relationships between faults and the error patterns that they cause); for specifying (e.g., relative to the source code of an application) where, when, and for how

long errors will strike; and how they will interact with executing object code or data. In its initial version, FIAT software can fault inject user application code and data and can inject faults into messages (corrupted, lost delayed), tasks (delayed, abnormal termination), and timers. Later versions will extend these fault injection capabilities into operating systems.

- XCEPTION: Developed at University of Coimbra, Portugal [6] uses the advanced bugging and performance monitoring features present in many of today's modern processors to inject fault. It also uses the processors own exceptions to trigger the faults. It requires no modification in application software and no insertion of software traps. The fault injector is implemented as an exception handler and requires modification of the interrupt handler vector. The Xception faults are trigger by access to specific addresses. This makes the experiments reproducible. Xception uses a fault mask when injecting a fault into a location in the system. The mask is compared with the memory/register/data and then the bits that are set to one in the mask are changed in the system by using bit-level-operations such as: Stuck-at-zero, stuck-at-one, bit-flip and bridging.
- DOCTOR: Integrated software fault injection environment developed at University of Michigan [16] allows injections into the CPU, memory and also network-communication faults. DOCTOR uses a more sophisticated method than the basic technique of modifying memory contents. Memory modification is a powerful fault injection method because almost every fault results, sooner or later, in some kind of contamination in the memory. Though it is a powerful method some faults may infect the memory in a very subtle and non-deterministic way, hence it can be very difficult to emulate such faults with basic memory modification. DOCTOR can use three different triggering mechanisms: Time-out triggered memory faults, when triggered the fault injector overwrites memory contents to emulate memory faults. Traps are used to create non-permanent CPU faults. For permanent CPU faults program instructions are changed during compilation to emulate instruction and data corruptions.
- EXFI: A fault injection system for embedded microprocessor-based boards developed at Politecnico di Torino, Italy [5]. The kernel of the EXFI system is based on the trace exception mode available in most microprocessors. During the fault injection experiment, the trace exception handler routine is in charge of computing the fault injection time, executing the injection of the fault, and triggering a possible time-out condition. The tool is able to inject single bit-flip transient faults both in the memory image of the process (data and code) and in the user registers of the processor. The approach can be easily extended to support different fault models, such as permanent stuck-at, couple, temporal and spatial multiple bit-flip, etc. The main characteristics of EXFI are the low cost (it does not require any hardware device), the high speed (which allows a higher number of faults to be considered), the low requirements in terms of features provided by the operating systems, the flexibility (it supports different fault types), and the high portability (it can be easily migrated to address different target systems).
- NFTAPE: Developed at the Center of Reliable and High Performance Computing at the University of Illinois at Urbana-Champaign [29]. The objective of NFTAPE is to support several different types of fault injection, providing the capability of targeting several heterogeneous systems concurrently. This is accomplished through use of a common control mechanism and common triggers. NFTAPE supports an arbitrary fault model. It can support a hardware fault injector to inject network faults, a SWIFI fault injector to inject communication faults, and a second SWIFI injector to target a distributed application. The first two injectors share an event-based trigger to coordinate communication faults, and the other uses a path-based trigger. Other fault injectors typically use one method of fault injection (say SWIFI or HWIFI), not to mention using multiple injectors at the same time or sharing triggers. In addition, NFTAPE contains a new driver based fault injection scheme, which unlike other SWIFI fault injectors, can inject faults into both kernel and user space with minimum required modifications for different operating systems.
- GOOFI (Generic Object-Oriented Fault Injection): Developed at the Department of Computer Engineering at Chalmers University of Technology in Sweden [1]. GOOFI can perform fault injection campaigns using different fault injection techniques on different target systems. A major objective of the tool is to provide a user-friendly fault injection environment with a graphical user interface and an underlying generic architecture that assists the user when adapting the tool for new target systems and new fault injection techniques. The GOOFI tool is highly portable between different host platforms since the tool was implemented using the Java programming language and all data is saved in a SQL compatible database. Furthermore, an object-oriented approach was chosen which increases the extensibility and maintainability of the tool. The current version of GOOFI supports pre-runtime Software Implemented Fault Injection (SWIFI) and Scan-Chain Implemented Fault Injection (SCIFI). The SCIFI technique injects faults via the built-in test-logic, i.e. boundary scan-chains and internal scan-chains, present in many modern VLSI circuits.

This enables faults to be injected into the pins and many of the internal state elements of an integrated circuit as well as observation of the internal state. In pre-runtime SWIFI, faults are injected into the program and data areas of the target system before it starts to execute. GOOFI is capable of injecting single or multiple transient bit-flip faults.

5. Simulation-Based Fault Injection

Simulation-based fault injection [18] involves the construction of a simulation model of the system under analysis, including a detailed simulation model of the processor in use. It means that the errors or failures of the simulated system occur according to predetermined distribution. The simulation models are developed using a hardware description language such as the Very high speed integrated circuit Hardware Description Language (VHDL). Faults are injected into VHDL models of the design and excited by a set of input patterns. It is important to note that VHDL constitutes a privileged language to comply with the goals of fault injection for the following reasons:

- Its widespread use in detailed design.
- Its inherent hierarchical abstraction description capabilities.
- Its ability to describe both the structure and behavior of a system in a unique syntactical framework.
- Its recognition as a viable framework for developing high-level models of digital systems.
- Its recognition as a viable framework for driving test activities.

An elementary fault injection *experiment* corresponds to one simulation run of the target system during which any number of faults can be injected on single or multiple locations of the model and at one or several points in time during the simulation. A series of experiments consists of a sequence of elementary fault injection experiments.

Several techniques have been proposed in the past to efficiently implement simulation-based fault-injection. Two main categories can be identified, those that require modification of VHDL code and those that use the built-in commands of the simulator. A first approach, based on *VHDL code modification*, modifying the system description by the addition of dedicated fault injection components called *saboteurs* or the mutation of existing component descriptions in the VHDL model which generates modified component descriptions called *mutants*. So that faults can be injected where and when desired, and their effects observed, both inside and on the outputs of the system.

A *saboteur* is a component added the VHDL model for the sole purpose of fault injection. It is inactive during normal system operation, while altering the value or timing characteristics of one or more signals when active, i.e when a fault is being injected. Saboteurs are inserted, in series or in parallel, either interactively at the schematic editor level or manually/automatically directly into the VHDL source code. Serial insertion, in its simplest form, consists of braking up the signal path between a driver (output) and its corresponding receiver (input) and placing a saboteur in between. In its more complex form, it is possible to break up the signal paths between a set of drivers and its corresponding set of receivers and insert a saboteur. For parallel insertion, a saboteur is simply added as an additional driver for a resolved signal (signal that have many drivers-signal sources – provided that a resolution function is supplied to resolve the values generated by the multiple sources into a single value). Saboteurs can be used to model most faults and to simulate environmental conditions such as noise or ESD. However, because they have no input pattern discrimination, saboteurs cannot model faults below the gate level of abstraction.

A *mutant* is a model which contains dormant code blocks within the normal gate description. These blocks of code are activated by injecting faults, altering the operation of the logic device itself. Because the fault response is generated internally within the model, any level of abstraction for fault injection is possible. However, the use of mutants requires that the original gate models be replaced by the new mutant models. This method main advantage is its complete independence on the adopted simulator, but it normally provides very low performance, due to the high cost for modification and possibly recompilation for every fault.

A second approach uses modified simulation tools (built-in commands of the VHDL simulators), which support the injection and observation features. This approach normally provides the best performance (does not require the modification of the VHDL code), but it can only be followed when the code of the simulation tools is available and easily modifiable, e.g., when fault injection is performed on zero-delay gate-level models. Its adoption when higher-level descriptions (e.g., RT-level VHDL descriptions) are used is much more complex. The applicability of these techniques depends strongly on the existing (commercial) simulators and on the functionality of their commands. Two techniques based on the use of simulator commands have been identified: VHDL signal manipulation (faults are injected by altering the value of the signals that are used to link the components that made up the VHDL model, this is done by disconnecting a signal from its driver(s) and forcing it to a new value) and VHDL variable manipulation (faults are injected into behavioral models by altering values of variables defined in VHDL processes).

A third approach relies on the simulation command language and interface provided by some specific simulator. The main advantage of this approach lies in the relatively low cost for its implementation, while the obtained performance is normally intermediate between those of the first and second approaches. It must be noted that it is now increasingly common for the new releases of most commercial simulation environments to support some procedural interface, thus allowing an efficient and portable interaction with the simulation engine and with its data structures. Several approaches have been presented for speeding up the simulation process. Fault injection techniques are compared in terms of fault modeling capacity, effort required for setting up an experiment and simulation time overhead.

Mutants offer the highest fault modeling capacity of the fault injection techniques presented, Saboteurs are generally less powerful, signal manipulation is suited for implementing simple fault models and variable manipulation offers a simple way for injecting behavioral faults.

The effort for setting up an experiment is small using signal and variable manipulation, as modification of the VHDL model is not required. More effort is needed for mutants and saboteurs (creation/generation, inclusion in the model, recompilation of the VHDL model).

The simulation time overhead imposed by signal and variable manipulation is only due to fault injection control, as the simulation must be stopped and started again for each fault injected. It is important to note that the simulation time overhead imposed by saboteurs and mutants depends on: Amount of additional generated events, amount of code to execute per event and the complexity of the fault injection control.

When considering a series of fault injection experiments, two ways can be distinguished: One way is to generate a new configuration for each fault location (this requires recompilation of the VHDL model for each fault location and may also require manual intervention to start up a simulation using the new model), another way is to generate only one configuration in which all required fault are included and then activate these one at a time (this may increase the simulation time). Thus, there is a trade-off between the overhead in simulation time and the overhead in compilation time.

Suppositions:

Model is an accurate representation of the actual system under analysis.

Benefits:

- Simulated fault injection can support all system abstraction levels-electrical, logical, functional, and architectural. It provides the maximum flexibility in terms of supported fault models.
- Not intrusive.
- Full control of both fault models and injection mechanisms.
- Low cost computer automation; does not require any special-purpose hardware.
- It provides timely feedback to system design engineers.
- Fault injection experiments are performed using the same software that will run in the field. Simulated fault injection can normally be rather easily integrated into already existing design flows.
- Maximum amount of observability and controllability. Essentially, given sufficient detail in the model, any signal value can be corrupted in any desired way, with the results of the corruption easily observable regardless of the location of the corrupted signal within the model. This flexibility allows any potential failure mode to be accurately modeled.
- Allows performing reliability assessment at different stages in the design process, well before than a prototype is available.
- Able to model both transient and permanent faults.
- Allows modeling of timing-related faults since the amount of simulation time required to inject the fault is effectively zero.

Drawbacks:

- Large development efforts.
- Time consuming (experiment length): Being based on the simulation of the system in its fault-free version as well as in the presence of the enormous number of the possible faults.
- Models are not readily available; rely on model accuracy
- Accuracy of the results depends on the goodness of the model used.
- No real time faults injection possible in a prototype.
- Model may not include any of the design faults that may be present in the real hardware.

Tools:

- VERIFY (VHDL-based Evaluation of Reliability by Injection Faults Efficiently): Developed at University of Erlangen-Nurnberg, Germany [27]. VERIFY uses an extension of VHDL for describing faults correlated to a component, enabling hardware manufacturers, which provide the design libraries, to express their knowledge of the fault behavior of their components. Multi-threaded fault injection which utilizes checkpoints and comparison with a golden run is used for faster simulation of faulty runs. The proposed extension to the VHDL language is very interesting but unfortunately

requires modification of the VHDL language itself. VERIFY uses an integrated fault model, the dependability evaluation is very close to that of the actual hardware.

- MEFISTO-C: A VHDL-based fault injection tool developed at Chalmers University of Technology, Sweden [12] that conduct fault injection experiments using VHDL simulation models. The tool is an improved version of the MEFISTO tool which was developed jointly by LAAS-CNRS and Chalmers. (A similar tool called MEFISTO-L has been developed at LAAS-CNRS). MEFISTO-C uses the vantage optimum VHDL simulator and injects faults via simulator commands in variables and signals defined in the VHDL model. It offers the user a variety of predefined fault models as well as other features to set-up and automatically conduct fault injection campaigns on a network of UNIX workstations.
- HEARTLESS: A hierarchical register-transfer-level fault-simulator for permanent and transient faults a simulator that was developed, by CE Group-BTU Cottbus in Germany, to simulate the fault behavior of complex sequential designs like processor cores [25]. Furthermore it serves for the validation of on-line test units for embedded processors. The input for HEARTLESS can support structural VHDL and ISCAS as input formats. It can support permanent stuck-at faults, transient bit flip and delay faults. HEARTLESS was developed in ANSI C++. The whole design or parts (macros) can be selected for fault simulation based on fault list generation. Fault-lits are collapsed according to special rules derived from logic level structure and signal traces. HEARTLESS can be enhanced by propagation over macros described in a C-function.
- GSTF: A VHDL-based fault injection tool developed by Fault Tolerance Systems Group at the Polytechnic University of Valencia, Spain [4]. This tool is presented as an automatic and model-independent fault injection tool to use on an IBM-PC or compatible system to inject faults into VHDL models (at gate, register and chip level). The tool has been build around a commercial VHDL simulator (V-System by Model Technology) and can implement the main injection techniques: Simulator commands, saboteurs and mutants. Both transient and permanent faults, of a wide range of types, can be injected into medium-complexity models. The tool can inject a wide range of fault models, surpassing the classical models of stuck-at and bit-flip and it is able to analyze the results obtained from the injection campaigns, in order to study the error syndrome of the system model and/or validate its fault-tolerance mechanisms.
- FTI (Fault Tolerance Injection): Developed at universidad Carlos III de Madrid in Spain, for fault-tolerant digital integrated circuits in the RT abstraction level [11]. The main objective of FTI is to generate a fault tolerant VHDL design description. Designer will provide an original VHDL design description and some guidelines about the type of fault-tolerant techniques to be used and their location in the design. FTI tool will process original VHDL descriptions by automatic insertion of hardware and information redundancy. Therefore, a unified format to deal with descriptions is needed. There are several intermediate formats that represent, by means of a database, the VHDL description in a formal way that could be accessed and processed with some procedural interface. Fault-tolerant components to be included into VHDL original descriptions will be already described and stored in a special library called FT library. These components come from previous researches about FT and designer just use them. FTI use an intermediate format for VHDL descriptions (FTL/TAURI) and it will work only with synthesizable descriptions IEEE 1076.
- [24, 28] Present a new techniques and a platform, developed at Politecnico di Torino – Italy, for accelerating and speeding-up simulation-based fault injection in VHDL descriptions and show how simulation time can be significantly shortened. The techniques developed analyze the faults to be injected in order to identify the final fault effects as early as possible and exploit the features provided by modern commercial VHDL simulators to speed-up injection operations. The ideas proposed in [23] was extended by making them more general and applying them dynamically during fault inject campaigns. The purpose of this approach is to minimize the time required for performing Fault Injection campaigns. This problem is addressed by performing fault analysis (before and during the Fault Injection campaign) and resorting to simulator commands that can be used to minimize the simulation time required to drive the system to the injection time. A prototypical version of the fault-injection platform has been devised in ANSI C, and consists of about 3,000 lines. Circuit analysis exploits FTL systems Tauri (a new version of the fault injector will be closely fastened to Auriga), fault-list generation takes advantage of Synopsis VHDL simulator, while the fault injector is currently based on Modelsim software.
- [10] Presents a fault injection technique, developed at Virginia University, USA, that allows faults to be injected at the ISA (Instruction Set Architecture) level where actual machine code is executed on a behavioral model of a processor written in VHDL. The idea of this technique is based on the use of a Bus Resolution Function (BRF) and the ability to communicate to the BRF when a fault is to be injected. This allow the BRF to corrupt the new value being assigned to a signal. A BRF is a

function associated with a signal type that will resolve the value of a signal declared to be of said signal type when the signal is being updated by two different sources at the same real time. This technique can be used with existing models with minimal changes to the existing code and it uses standard VHDL types to perform the fault injection (it is simulator-independent method). The simulation time is reduced because the level of modeled detail is reduced. However, this method is limited to processor fault-injection modeled in the ISA level.

6. Emulation-Based Fault Injection

To cope with the time limitations imposed by simulation and take into account the effects due to the circuit environment in the application, in system emulation using hardware prototyping on FPGA-based logic emulation systems has been proposed [9, 20]. The circuit to analyze is implemented onto the FPGA using a classical synthesis, placement and routing design flow starting from the high-level circuit description. The development board is connected to a host computer, used to define the fault injection campaign, control the injection experiments and display the results.

In some limited cases, the approaches developed for fault grading using emulators (for example [7]) may be used to inject faults. However, such approaches are classically limited to stuck-at fault injection. In most cases, modifications must therefore be introduced in the circuit description taking into account that the description must remain synthesizable and satisfying a set of constraints related to the emulator hardware. The modifications are therefore not easy and furthermore it is often necessary to generate several modified descriptions, each of them allowing the injection of a given subset of faults. In such a case, the hardware emulator has in general to be completely reconfigured several times, that is quite time-consuming and reduces the gain in execution time compared with simulation. It also implies additional synthesis, place and route phases since the whole design flow has to be executed for each modified description.

FPGAs have already been used to accelerate fault-injection in a number of cases. In general, these approaches aim at using the high running speed of a hardware prototype to reduce the fault injection experiment time with respect to simulations. New methodologies were also introduced combining hardware-based and software-based techniques in order to exploit the speed of hardware-based techniques and at the same time take profit of the flexibility of software-based techniques. In general, additional control inputs and specific elements are introduced by modifying either the initial high-level circuit description or the gate-level description so that the targeted faults can be injected into the prototype. This is sometimes called "instrumenting" the circuit description. As previously mentioned, the emulator characteristics can preclude generating a single instrumented description allowing injecting all the targeted faults. This may be due to the limited number of available I/Os, or to the amount of hardware overhead induced by the logic elements added in the circuit for fault injection. In that case, each version of the instrumented description targets a given subset of faults and has to be separately synthesized, placed, routed and downloaded onto the emulator at different phases of the injection campaign.

To avoid any instrumentation of the circuit description, another approach, called *run-time reconfiguration emulation-based fault injection* has proposed in [21]. Instead of injecting the faults by means of specific external signals controlling additional logic, these approaches rely on built-in reconfiguration capabilities of the FPGA devices. This means that some run-time reconfiguration has to be done for each fault to inject; however, this avoids the extra time spent in preparing the instrumented versions. The bit stream modification necessary to perform the reconfigurations is a very quick process compared for example with synthesis. Also, the reconfiguration time globally spent when running a fault injection campaign on the hardware emulator (FPGA) can be reduced by means of a partial reconfiguration of the emulator when such capabilities are available.

The initial VHDL description is therefore synthesized, placed & routed and a bit file is generated, corresponding to the targeted circuit without any additional elements. The generated file is downloaded onto the FPGA and the injection campaign begins by an execution of the studied workload (or test bench) on the implemented prototype. The result of this execution is later used as reference for analyzing the effects of faults. Then, the same workload is run again as many times as there are faults (or fault configurations) to inject. Run-Time Reconfiguration (RTR) had been proposed as a technique to inject the faults. This methodology propose to inject the faults at "low-level", directly in the reconfigurable hardware, by modification of the design previously implemented in the FPGA. So any fault injection can be realized without changing the initial description and without additional hardware. The first advantage is to avoid any hardware overhead for fault injection, that may allow the designer to perform the emulation on a smaller FPGA. Also, carrying out the modifications directly in the reconfigurable device can only take a fraction of a second if partial reconfiguration can be achieved. So noticeable time gains can be expected with respect to "classical" fault injection techniques, although a reconfiguration is required for each fault configuration to inject. Then an extra time is needed in

each fault injection experiment, as a partial read back and a partial reconfiguration is needed to inject a fault. This extra time could be however relatively low compared with a classical simulation cycle time.

Noticeable gains could be expected compared with simulation-based injection experiments, provided that the configuration of the FPGA is quick enough. This implies to optimize several implementation characteristics:

- Intrinsic reconfiguration time of the reconfigurable device (related to its architecture and to the place and route algorithms used); a good solution would be to use a device not only with partial reconfigurability but also with some kind of random access to the configuration data
- High configuration bandwidth on the development board (high frequency configuration clock and/or configuration data sent in parallel mode onto the FPGA)
- High bandwidth interface between the development board and the host computer.

In conclusion, let us summarize the advantages and disadvantages of this technique.

Benefits:

- Injection time is more quickly compared with simulation-based techniques possibility of in-system emulation, allowing the designer to evaluate much more precisely the behavior which can be expected in the final circuit environment.
- Would especially be interesting in the context of a system-on-chip development since it may lead to efficient but low cost dependability analysis of re-usable components (most often called IP blocks), before they are used in a given circuit.
- The experimentation time can be reduced by implementing partially or totally the input pattern generation in the FPGA. These patterns are already known when the circuit to analyze is synthesized.

Drawbacks:

- The initial VHDL description must be synthesizable and optimized to avoid requiring a too large and costly emulator and to reduce the total running time during the injection campaign.
- The cost of a general hardware emulation system and/or the implementation complexity of a dedicated FPGA based emulation board. A low cost can be reached but at the expense of a reduced speed of the injection fault campaign.
- The emulation is only used to analyze the functional consequences of a fault, the temporal impacts of the faults are not considered. They are looking only at steady states of the signals at some particular moments (in general just before the rising and/or falling edge of the clock).
- Since the algorithmic description are not yet widely accepted by synthesis tools in classical industrial design flows, the approach using the emulation can often only be applied starting from RT-level descriptions.
- I/Os problems: When using a FPGA-based development board, the main limitation becomes the number of I/Os of the programmable hardware, which can be connected between the FPGA and the host computer, that can restricts the number of fault injection signals and the number of monitored signals.
- Necessity of high speed communication link between the host computer and the emulation board: This is the actual critical part of the emulation set-up.

7. Hybrid Fault Injection

A hybrid approach combines two or more of the other fault injection techniques to more fully exercise the system under analysis. For instance, performing hardware-based or software-based fault injection experiments can provide significant benefit in terms of time to perform the fault injection experiments, can reduce the initial amount of setup time before beginning the experiments, and so forth. The hybrid approach combines the versatility of software fault injection and the accuracy of hardware monitoring. The hybrid approach is well suited for measuring extremely short latencies. However, given the significant gain in controllability and observability with a simulation-based approach, it might be useful to combine a simulation-based approach with one of the others in order to more fully exercise the system under analysis. For instance, most researchers and practitioners might choose to model a portion of the system under analysis, such as the Arithmetic and Logic Unit (ALU) within the microprocessor, at a very detailed level, and perform simulation-based fault injection experiments due to the fact that the internal nodes of an ALU are not accessible using a hardware-based or software-based approach.

Tools:

- LIVE: Experimental evaluation of computer-based railway control systems, developed at Ansaldo-Cris, Italy integrates fault injection and software testing techniques to achieve an accurate and non-intrusive analysis of a system prototype [2]. It uses pin-level forcing or generates interrupts to activate software fault injection procedures. A method combining software-based and simulation-based fault injection developed at Chalmers University of Technology, Sweden [15]. This hybrid fault injection technique, also known as mixed-mode fault injection, allows the advantages of both SWIFI (Software

Implemented Fault Injection tool) and simulation based fault injection to be utilized, i.e. the actual target system may be executed at full speed except during the injection of a fault when a simulator providing detailed access to the target system is used instead. The technique is combined with operational-profile-based fault injection which only injects faults in those parts (e.g. registers) which contain live data, i.e. which will not be overwritten.

8. Conclusions

The last years marked growing demand for new techniques to be applied in the design of fault tolerant electronic systems, and for new tools for supporting the designers of these systems. The increased interest for the domain of fault tolerant electronic systems design stems primarily from the extension in their use to many new areas. At the same time, the cost and time-to-market minimization constraints obviously affect the design of fault tolerant systems, and new techniques and new tools are continuously needed to face these constraints.

Fault injection is an important technique for the evaluation of design metrics such as reliability, safety and fault coverage. Fault injection involves inserting faults into a system and monitoring the system to determine its behavior in response to the fault.

In this paper we have described several techniques that have been made to develop techniques for injecting fault into a system prototype or model. These techniques fall into five categories: Hardware-based fault injection, software-based fault injection, simulation-based fault injection, emulation-based fault injection and hybrid fault injection. In table 1, we summarize the main advantages and disadvantages of these techniques.

Most recent research in this area is converging towards hybrid fault injection combining the benefits of both hardware and software fault injection techniques, while avoiding most of their disadvantages. This is becoming feasible due to the latest advancements in the FPGA technology. Modern FGPA devices can be fruitfully exploited to emulate systems composed of hundreds of thousands of gates at a reasonable cost.

Table 1. Summary of main advantages and disadvantages of fault injection techniques.

Techniques	Advantages	Disadvantages
Hardware-Based	• Can access locations that is hard to be accessed by other means. • High time-resolution for hardware triggering and monitoring. • Well suited for the low-level fault models. • Not intrusive. • Experiments are fast. • No model development or validation required. • Able to model permanent faults at the pin level.	• Can introduce high risk of damage for the injected system. • High level of device integration, multiple-chip hybrid circuit, and dense packaging technologies limit accessibility to injection. • Low portability and observability. • Limited set of injection points and limited set of injectable faults. • Requires special-purpose hardware in order to perform the fault injection experiments.
Software-Based	• Can be targeted to applications and operating systems. • Experiments can be run in near real-time. • Does not require any special-purpose hardware; low complexity, low development and low implementation cost. • No model development or validation required. • Can be expanded for new classes of faults.	• Limited set of injection instants. • It cannot inject faults into locations that are inaccessible to software. • Does require a modification of the source code to support the fault injection. • Limited observability and controllability. • Very difficult to model permanent faults.
Simulation-Based	• Can support all system abstraction levels. • Not intrusive. • Full control of both fault models and injection mechanisms. • Low cost computer automation; does not require any special-purpose hardware. • Maximum amount of observability and controllability. • Allows performing reliability assessment at different stages in the design process. • Able to model both transient and permanent faults.	• Large development efforts. • Time consuming (experiment length). • Model is not readily available. • Accuracy of the results depends on the goodness of the model used. • No real time faults injection possible in a prototype. • Model may not include any of the design faults that may be present in the real hardware.
Emulation-Based	• Injection time is more quickly compared with simulation-based techniques. • The experimentation time can be reduced by implementing partially or totally the input pattern generation in the FPGA. These patterns are already known when the circuit to analyze is synthesized.	• The initial VHDL description must be synthesizable and optimized to avoid requiring a too large and costly emulator and to reduce the total running time during the injection campaign. • The cost of a general hardware emulation system and/or the implementation complexity of a dedicated FPGA based emulation board. • The emulation is only used to analyze the functional consequences of a fault. • When using a FPGA-based development board, the main limitation becomes the number of I/Os of the programmable hardware. Necessity of high speed communication link between the host computer and the emulation board.

Acknowledgements

This work is supported by the Lebanese University research program and the French/Lebanon CEDRE program. We would like to acknowledge the support of all these organizations for their help and contributions.

References

[1] Aidemark J., Vinter J., Folkesson P., and Karlsson J., "GOOFI: Generic Object-Oriented Fault Injection Tool," *in Proceedings of International Conference on Dependable Systems and Networks (DSN'2001)*, Gothenburg, Sweden, July 2001.

[2] Amendola A., Impagliazzo L., Marmo P., and Poli F., "Experimental Evaluation of Computer-Based Railway Control Systems," *in Proceedings of 27th International Symposium on Fault-Tolerant Computing (FTCS-27)*, Seattle, WA, USA, pp. 380-384, June 1997.

[3] Arlat J., "Validation de la Sûreté de Fonctionnement Par Injection de Fautes. Méthode Mise en Œuvre et Application," *Thèse Présentée à l'INP Toulouse*, Rapport de Recherche LAAS, no. 90-399, December 1990.

[4] Baraza J. C., Gracia J., Gil D., and Gil P. J., "A Prototype of a VHDL-Based Fault Injection Tool," *in Proceedings of DFT'2000 Conference*, pp. 396-404, October 2000.

[5] Benso A., Prinetto P., Rebaudengo M., and Reorda M., "EXFI: A Low-Cost Fault Injection System for Embedded Microprocessor-Based Boards," *ACM Transactions on Design Automation of Electronic Systems*, vol. 3, no. 4, pp. 626-634, October 1998.

[6] Carreira J., Madeira H., and Silva J., "Xception: A Technique for the Experimental Evaluation of Dependability in Modern Computers," *IEEE Transactions on Software Engineering*, vol. 24, no. 2, pp. 125-136, February 1998.

[7] Cheng K. T., Huang S. Y., and Dai W. J., "Fault Emulation: A New Methodology for Fault Grading," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 18, no. 10, pp. 1487-1495, October 1999.

[8] Choi G. S. and Iyer R. K., "FOCUS: An Experimental Environment for Fault Sensitivity Analysis," *IEEE Transactions on Computers*, vol. 41, no. 12, pp. 1515-1526, December 1992.

[9] Civera P., Macchiarulo L., Rebaudengo M., Reorda S. M., and Violante M., "Exploiting FPGA for Accelerating Fault Injection Experiments," *in Proceedings of 6th International On-Line Testing Workshop*, Palma de Mallorca, Spain, July 2000.

[10] Delong T. A., Johnson B. W., and Profetan J. A., "A Fault Injection Technique for VHDL Behavioral-Level Models," *IEEE Design & Test of Computers*, vol. 13, pp. 24-33, 1996.

[11] Entrena L., López C., and Olías E., "Automatic Generation of Fault Tolerant VHDL Designs in RTL," *FDL (Forum on Design Languages)*, Lyon, France, September 2001.

[12] Folkesson P., Svensson S., and Karlsson J., "A Comparison of Simulation Based and Scan Chain Implemented Fault Injection," *in Proceedings of 28th International Symposium on Fault-Tolerant Computing (FTCS-28)*, Munich, Germany, pp. 284-293, June 1998.

[13] Fuchs E., "An Evaluation of the Error Detection Mechanisms in MARS Using Software Implemented Fault Injection," *in Proceedings of 2nd European Dependable Computing Conference (EDCC-2)*, Taormina, Italy, October 1996.

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

[15] Guthoff J. and Sieh V., "Combining Software-Implemented and Simulation-Based Fault Injection into a Single Fault Injection Method," *in Proceedings of 25th Symposium on Fault-Tolerant Computing (FTCS)*, pp. 196-206, 1995.

[16] Han S., Rosenberg H., and Shin K., "DOCTOR: An Integrated Software Fault Injection Environment," *Technical Report CSE-TR-192-93*, University of Michigan, 1993.

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

[18] Jenn E., Rimen M., Ohlsson J., Karlsson J., and Arlat J., "Design Guidelines of a VHDL-Based Simulation Tool for the Validation of Fault Tolerance," *in Proceedings of 1st ESPRIT Basic Research Project PDCS-2 Open Workshop*, LAAS/CNRS, Toulouse, pp. 461-483, September 1993.

[19] Kanawati G. A., Kanawati N. A., and Abraham J. A., "FERRARI: A Tool for the Validation of System Dependability Properties," *in Proceedings of 22nd Annual International Symposium Fault-Tolerant Computing*, IEEE CS Press, Los Alamitos, California, pp. 336-344, 1992.

[20] Leveugle R., "Fault Injection in VHDL Descriptions and Emulations," *in Proceedings of DFT'2000 Conference*, pp. 414-419, October 2000.

[21] Leveugle R., Antoni L., and Feher B., "Using Run-Time Reconfiguration for Fault Injection in Hardware Prototypes," *in Proceedings of DFT'2000 Conference*, November 2002.

[22] Madeira H., Rela M., Moreira F., and Silva J. G., "RIFLE: A General Purpose Pin-level Fault

Injector," *in Proceedings of 1st European Dependable Computing Conference (EDCC-1)*, (Berlin, Germany), Springer-Verlag, pp. 199-216, 1994.

[23] Rebaudengo M., Benso A., Marmo P., and Impagliazzo L., "Fault-List Collapsing for Fault Injection Experiments," *in Proceedings of Annual Reliability and Maintainability Symposium (RAMS'98)*, pp. 383-388, January 1998.

[24] Rebaudengo M., Prrota B., Violante M., and Sonza R. M., "New Techniques for Accelerating Fault Injectioning VHDL Description," *in Proceedings of International On-Line Test Workshop (IOLTW'2000)*, Mallorca, Spain, pp. 61-66, July 2000.

[25] Rousselle C., Pflanz M., Behling A., Mohaupt T., and Vierhaus H. T., "A Register-Transfer-Level Fault Simulator for Permanent and Transient Faults in Embedded Processors," *in Proceedings of DATE'2001 Conference*, Munich, Germany, 2001

[26] Segall Z., Vrsalovic D., Siewiorek D., Yaskin D., Kownacki J., Barton J., Dancey R., Robinson A., and Lin T., "FIAT-Fault Injection Based Automated Testing Environment," *in Proceedings of 18th International Symposiumon Fault-Tolerant Computing (FTCS-18)*, pp. 102-107, 1988.

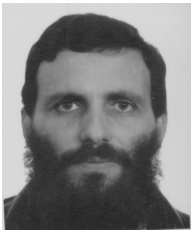
[27] Sieh V., Tschäche O., and Balbach F., "VERIFY: Evaluation of Reliability Using VHDL-Models with Embedded Fault Descriptions," *in Proceedings of 27th International Symposiumon Fault-Tolerant Computing (FTCS-27)*, Seattle, WA, USA, pp. 32-36, June 1997.

[28] Sonza Reorda M., Berrojo L., González I., and Corno F., "New Techniques for Speeding-up Fault-Injection Campaigns," *in Proceedings of (DATE'2002) Design Automation and Test in Europe*, Paris, France, pp. 847-852, March 2002.

[29] Stott D. T., Kalbarczyk Z., and Iyer R. K, "Using NFTAPE for Rapid Development of Automated Fault Injection Experiments," *Research Report*, Center for Reliable and High-Performance Computing, University of Illinois at Urbana-Champaign, 1999.

[30] Tsai T. K. and Iyer R. K., "An Approach to Benchmarking of Fault-Tolerant Commercial Systems," *in Proceedings of 26th Annual International Symposium Fault-Tolerant Computing*, IEEE CS Press, Los Alamitos, California, pp. 314-323, 1996.

[31] Yu Y., "A Perspective on the State of Research on Fault Injection Techniques," *Research Report*, University of Virginia, May 2001.



Haissam Ziade received his BSc in physics from Lebanese University in 1979, his engineering diploma from ENSERG in Grenoble France in 1982, and his PhD in engineering from INSA/ Toulouse in 1986. Since 1986, he has been in the Electrical and Electronics Department at the Lebanese University at Tripoli/Lebanon, where he is currently an associate professor. He is an assistant researcher at TIMA Laboratory (Grenoble, France) in "Qualification of Circuits" research group since 1986. His main research topics are the study of the test and validation of complex integrated circuits, the fault injection methodologies and the design with programmable circuits and systems.

Rafic Ayoubi received his BSc degree in electrical engineering, the MSc and PhD degrees in computer engineering from the University of Louisiana, Lafayette, Louisiana in 1988, 1990, and 1995, respectively. He joined The University of Balamand, Tripoli, Lebanon, in 1996 where he is currently an assistant professor. Dr. Ayoubi's current research interests are parallel architectures, parallel algorithms, fault tolerance, artificial neural networks, and FPGA technology. In these areas, he published several research papers in several journals and conferences. He has received the first prize in the 2nd Annual Exhibition for Industrial Research Achievements in Lebanon.

Raoul Velazco has been with CNRS (French Research Agency) since 1984. Leader at TIMA Laboratory (Grenoble, France) of "Qualification of Circuits" research group. His main research topics are the study of the effects of radiation on integrated circuits, the development of test methods for complex circuits (processors, microcontrollers,...) and the design of dedicated functional test systems. He has more than 170 publications, 32 of them in IEEE Transactions on Nuclear Science.