Using Run-Time Reconfiguration for Fault Injection in Hardware Prototypes

Lörinc Antoni[*+], Régis Leveugle[*], Béla Fehér[+]

[*]TIMA Laboratory
46, Avenue Félix Viallet
38031 GRENOBLE Cedex, France

[+]Budapest University of Technology and Economics
Department of Measurement and Information Systems
H-1111 Budapest, Műegyetem rkp. 9. Bldg. R, I. 113.

E-mails: [Lorinc.Antoni, Regis.Leveugle]@imag.fr, feher@mit.bme.hu

Abstract

In this paper, a new methodology for the injection of Single Event Upsets (SEU) in memory elements is introduced. SEUs in memory elements can occur due to many reasons (e.g. particle hits, radiation) and at any time. It becomes therefore important to examine the behaviour of circuits when a SEU occurs in them. Reconfigurable hardware (especially FPGAs) has shown to be suitable to emulate the behaviour of a logic design and to realise fault injection. The proposed methodology for SEU injection exploits FPGAs and, contrarily to the most common fault injection techniques, realises the injection directly in the reconfigurable hardware, taking advantage of run-time reconfiguration capabilities of the device. In this case, no modification of the initial design description is needed to inject a fault, that results in avoiding hardware overheads and specific synthesis, place and route phases.

1: Introduction

Several types of tests can be applied to a circuit once the basic validations are successful. The fault injection techniques have been recognised for a long time as necessary to validate the dependability of a system by analysing the behaviour of the devices when a fault occurs. Also, reconfigurable hardware is appropriate to implement and test prototypes by synthesizing descriptions written in high-level languages such as VHDL. Another advantage of prototyping is the possibility to perform ”in-system” emulation before any manufacturing.

Run-Time Reconfiguration (RTR) is a well-known technique that reconfigures hardware during execution of the application. There are different kinds of RTR depending on whether all the circuitry is reconfigured or only part of it, and if it is only partially reconfigured then the rest of the circuit either remains in operation or not.

The features of reconfigurable hardware described above make it suitable to implement prototypes in which fault injections are performed. In [2] and [3] methodologies have been proposed and validated to realise fault injections using a run-time reconfiguration of a circuit prototype. These approaches allowed us to inject stuck-at faults or bit-flips in the combinatorial parts of a circuit. The injected faults could be permanent or transient ; in the latter case, a second reconfiguration of the prototype has to be performed to suppress the fault after a given number of cycles. In this paper an approach based on similar principles is proposed to achieve the injection of bit-flips directly in memory elements, asynchronously with respect to the system clock. This corresponds therefore exactly to the occurrence of Single Event Upsets (SEUs).

The paper is organised as follows. Section 2 contains a brief review of the most common fault injection techniques. The type of methodology discussed in this paper is compared more precisely to the existing techniques in Section 3. The new methodology proposed in this paper is then detailed in section 4. The results of the experiments are summarised in section 5 and conclusions can be found in section 6.

2: Most common fault injection techniques

As previously mentioned, fault injection techniques have been proposed for a long time to evaluate the dependability of a given circuit or system implementation. Most of the approaches proposed up to now apply once the system or circuit is available. Such approaches include pin-level fault injection, memory corruption [26] [17] [16], heavy-ion injection [27] [21], power supply disturbances [20], laser fault injection [29] or software fault injection [18] [19] [6] [4] [5].

More recently, several authors proposed to apply fault injection early in the design process. The main approach consists in injecting the faults in high level models (most often, VHDL models) of the circuit or system. [13] describes for example the injection of faults in behavioural VHDL descriptions of microprocessor-based systems. [15], and then [30] or [8], consider the injection of different types of faults in the VHDL model of a circuit at several abstraction levels and using various techniques based on the modification of the initial VHDL description. As in the case of [13], simulations are used to evaluate the impact of the faults on the circuit behaviour. As mentioned in [23], the main drawback related to the use of simulations is the huge amount of time required to run the experiments when many faults have to be injected in a complex circuit.

To cope with the time limitations imposed by simulation, it has been proposed to take advantage of hardware prototyping, using a FPGA-based hardware emulator [22]. Another advantage of emulation is to allow the designer to study the actual behaviour of the circuit in the application environment, taking into account real-time interactions [7]. When an emulator is used, the initial VHDL description must of course be synthesizable. In some limited cases, the approaches developed for fault grading using emulators (e.g. [9], [32]) may be used to inject faults. However, such approaches are classically limited to stuck-at fault injection. In the special case of the SimExpress emulator, faults can also be injected in the circuit prototype by using built-in facilities of the emulator [1]. Nevertheless, such facilities do not exist today in the other emulators commercially available. Furthermore, such an injection is limited to the single stuck-at fault model. 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 [23]. 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.

3: Position of the type of approach proposed in this paper

FPGAs have already been used to accelerate fault-injection in a number of cases [23] [11] [10] [12] [25]. 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 in [10] [12] 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 [24] or the gate-level description [11] 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 to inject 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.

The main goal of the type of approach proposed in [2] [3] and in this paper is to avoid any instrumentation of the circuit description. 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 bitstream modifications 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.

In the type of approach considered in this paper, the initial VHDL description is therefore synthesized, placed & routed and a bitfile 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 testbench) on the implemented prototype. The result of this execution is later used as reference for analysing the effects of faults. Then, the same workload is run again as many times as there are faults (or fault configurations) to inject. In [2] and [3], Run-Time Reconfiguration (RTR) had been proposed as a technique to inject the faults. These publications dealt with the injection of stuck-at faults or bit-flips in the combinatorial parts of a circuit by reconfiguration of the Look-Up Tables (LUT) of the FPGA. The injected faults could be permanent or transient ; in the latter case, a second reconfiguration of the prototype had to be performed to suppress the fault after a given number of cycles.

In the methodology described in this paper, injection of SEUs is proposed by directly changing the state of the memory elements in an asynchronous way. It means that the execution of the application running on the FPGA device is stopped, the SEUs are injected by changing the state of one or more flip-flops and then, the execution of the application is continued.

4: The proposed methodology

Most of the fault injection techniques described in the previous sections are applied at ”high-level” in the design flow, generally working on the VHDL description. Faults are injected by modifying this VHDL description, that should be re-worked to be implemented onto the reconfigurable hardware. The additional synthesis, place & route and bitstream generation phases can take from a few minutes to days, depending on the size of the design.

The methodology proposed hereby injects the faults at ”low-level”, directly in the reconfigurable hardware, by modification of the design previously implemented in the FPGA. 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.

4.1: General description

The first step in the methodology is the specification and functional validation of the circuit (like in all other fault injection techniques). In our case, the design can be made indifferently by schematic capture or by using a description language (e.g. VHDL). After synthesis (when required), mapping and place & route, a bitstream is generated that can be downloaded onto the reconfigurable hardware (e.g. FPGA). In the proposed methodology, it is only this bitstream that is modified in order to inject faults, so no more synthesis, mapping and place & route is needed after the first bitfile generation.

In our experiments, we use Virtex devices from Xilinx. Once the bitstream is available, it is possible to make the modifications on it by a Java-based application programming interface called JBits. The JBits API is described in section 4.2. This tool set not only lets us possible to make modifications directly in the bitstream, but also to reconfigure the bitstream directly in the FPGA device at run-time. Moreover, JBits makes possible to carry out partial reconfiguration and readback of the device. Like this, fault injection can easily be realised as it is described in section 4.3.

4.2: JBits

The JBits API is a Java-based tool set, or application programming interface (API), that allows designers to write information directly to a Xilinx FPGA to carry out whatever customer logic operations were designed for it [14] [34]. The JBits API permits the FPGA bitstream to be modified quickly, allowing for fast reconfiguration of the FPGA. With Virtex FPGAs, the JBits API can partially or fully reconfigure the internal logic of the hardware device.

The Virtex architecture allows this reconfiguration to be as extensive as necessary and still maintain timing information [14] [34]. The JBits API also makes it possible to integrate the operations of the FPGA with other system components such as an embedded processor, a graphics coprocessor, or any digital peripheral device.

Some applications are known that have been developed with the JBits tool. Up to our knowledge, we made the first attempts to use Jbits to perform fault injection.

4.3: The fault injection program and the platform

Injection of an SEU means normally that the state of a memory element changes to the opposite. The proposed approach focuses on the SEUs in the flip-flops (FFs) of the FPGA device used as functional memory elements of the circuit under analysis. The basic element of the Xilinx FPGAs are the Configurable Logic Blocks (CLB) that are divided into two identical slices in case of Virtex FPGAs. The application realising the injection of SEUs has been developed in Java, using the JBits API (Section 4.2). The main idea was to inject SEUs by directly changing the state of the FFs in the CLB slices.

The only way of changing asynchronously the state of a FF is to apply a set or reset to it, depending on whether it should be changed to 1 or 0. When applying run-time reconfiguration with JBits, it is possible only to pulse the Global Set/Reset (GSR) line of the device, that sets/resets all the FFs of the device depending on the function of the GSR line. This function is determined for each flip-flop with a switch. Fig. 1 shows a very simplified view of a FF and the GSR line in the CLB of a Virtex FPGA.

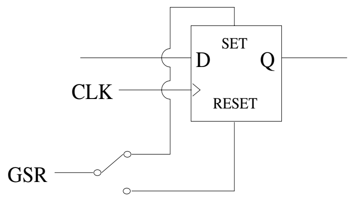

Figure 1. Simplified figure of a flip-flop in the CLB of a Virtex FPGA

In this case, if one should change the state of only one FF (that is the case for SEU injection), the states of all FFs and set/reset switches must be read first. If there are FFs where the switch is not in the same state as the FF (e.g. the switch is in "set" but the FF state is "0"), one must change the switch before pulsing the GSR line in order to leave the FF in its initial state. On the other hand, in the case of the FF where the SEU should be injected, the switch must be set in the opposite state of the FF state.

The basic steps of the algorithm used for asynchronous SEU injection are thus as follows:

- Initialisation: reading of the bitstream and full configuration of the device
- Read the states of set/reset switches

Then for each injection experiment:

- Start execution of the application
- Stop the execution at injection cycle
- Read the states of FFs
- Change the set/reset switches where needed (depending on FF states)
- Pulse the GSR line
- Re-change the switches modified before GSR
- Continue the execution of the application until the end of the experiment

To change the state of either a FF or a switch, the FPGA device must be reconfigured. As it is mentioned in section 4.2, it is possible to partially reconfigure or read back the device by using JBits, in case of FPGA families like Virtex. Like this, when changing/reading back FF or switch states, not all the device must be changed/read back but only a part of it.

Of course, before beginning the execution of the application, the bitstream containing the synthesized, placed and routed design must be first read from a file, and the FPGA device must be configured with this bitfile by a full reconfiguration. Also, in addition to the partial reconfigurations for injecting the faults, some FFs (or all of them) may have to be read back, to capture their states during execution at each clock cycle or at given times in the experiments. These states can be compared with the reference FF states obtained when executing the application without faults, in order to compare the behaviour of the application with faults and without faults. The read back of the FF states is either a full or better a partial read back of the device. Depending on the type of analysis required by the user, the read backs can therefore be a non negligible part of the time overhead during the experiments, much higher than the injection time ; using partial read back is therefore very important to reduce the length of the injection campaign.

During execution of a fault injection campaign, not only faults are injected into the circuit, but of course the test patterns corresponding to the workload have to be applied to the inputs of the circuit. This can be done in the easiest way by memories, in which the input vectors are stored and that are implemented onto the FPGA along with the analysed circuit. This avoids time consuming data exchanges between the host computer and the hardware prototype ; these exchanges can be limited to reading and storing the design responses. A disadvantage of this method is that input data memories can occupy a non negligible space in the reconfigurable hardware. However, with novel FPGA families that have got a capacity up to 8 million gates, this does not really cause problems in most cases. Also, partial reconfiguration can be used here again to limit the size of the implemented memory. In this case, the memory contents is reconfigured a given number of times during an experiment, that leads to a trade-off between the experiment length and the area occupied in the reconfigurable hardware by the input vectors.

To summarise, there are two very important differences between our methodology and the other ones described in Sections 2 and 3. First of all, in other techniques the high-level or gate-level description must be changed in order to be able to inject the faults. In our approach, fault injection is realised at ”low-level”, so any fault injection can be realised without changing the initial description and without additional hardware. The second difference is that in our case 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, as it will be discussed in Section 5.

5: Results

5.1: Device-based analysis

The analyses made during the reported work to validate the proposed concept were carried out on the example of a XCV50 device. This Virtex component is made of a 16x24 matrix of CLBs and can be configured either in parallel (8-bit words) or in series at a frequency of up to 60 MHz. The flip-flops can be read-back and the switches can be configured in each of the 24 columns using four 384-bit frames. Using partial reconfiguration, one can expect that only some of these frames have to be read back or reconfigured during each injection experiment. However, the number of frames to consider in the proposed approach does not only depend on the number of flip-flops actually used in the design or on the position of the switches at injection times ; it also depends on the placement of these flip-flops in the CLBs of the device. In the best case, all the functional flip-flops are in a small subset of the columns, or grouped into blocks corresponding to the device hierarchy, and a limited number of frames has to be considered. In the worst case, the flip-flops to be read or modified are spread over all the columns and all the blocks, requiring to define all the frames, and therefore requiring a read back and a reconfiguration of all flip-flops, no matter the number of flip-flops actually used in the design. With this device architecture, it is therefore clear that the efficiency of the proposed approach is dependent on the placement and routing algorithms. Only some kind of random access to the configuration data would avoid this type of dependence.

Let us consider a configuration clock (CCLK) at 1 MHz. In the worst case, for the XCV50 device, 36,864 bits must be read back or configured during one basic step of our injection methodology. In the case of a serial configuration, this leads to a partial read back or configuration time of up to 37 ms. The time overhead due to the injection process can thus be evaluated around 100 ms on an average, including the CPU time required by the bitstream modifications. This was confirmed by preliminary experiments on very simple circuit examples, showing injection times between 50 and 150 ms. These results have to be compared with times of a few seconds for a full reconfiguration of the device (configuration of 1450 frames). Such a full reconfiguration is only done once in our methodology, when initialising the injection campaign.

A simple evaluation based on the device characteristics shows that much better performances could be achieved using a parallel transfer of the configuration data and the maximum configuration frequency. In this case, the injection overhead could be less than 1 ms per experiment.

The advantages and limits of the presented approach can be deduced from such simple data. It is easy to deduct some estimations regarding any design, as the complexity of the circuit under analysis only has a limited impact on the injection times. In practice, the placement of the flip-flops and the number of switches to commute for a given injection are more significant parameters than the design complexity in terms of gates. Also, the system parameters (e.g. the frequency of the configuration clock) have a stronger impact than the design complexity.

On the basis of an average 100 ms overhead per fault injection, the time required during a campaign to inject 1000 faults is less than 2 minutes, that is not really significant compared to the total experiment duration, and the time increase is linear with the number of faults to inject. Looking more closely to this evaluation, comparisons can be made between the proposed approach and two previous ones: the simulation-based approach and the emulation-based approach using an instrumented description of the circuit. In this latter case, it will be assumed that the injection only involves changing the state of additional external control signals and therefore does not induce any additional execution cycle. We will also not take into account the initialisation times for the simulator or the emulator, and we will assume that a potential failure is identified by only looking at the results once at the end of each experiment, leading to a negligible time overhead. Of course, the comparisons may have to be revisited for all the other cases, but the main conclusions presented here should still hold.

To compare the proposed approach with simulation-based injection, we made experiments using the ModelSim VHDL simulator from Mentor Graphics, executed on a Sun UltraSparc workstation. The clock frequency on the prototype was assumed to be only 10 MHz, that may be improved in most practical cases. Some comparisons are shown in Table 1, based on extrapolations of the measures obtained during our experiments. Looking at a small circuit with 100 memory points and a testbench using 1,000 input patterns (or 1,000 simulation cycles), the number of injected faults to cover all the potential cases is 100,000. In such a case, the emulation using the proposed methodology can achieve a gain of about 58% in experimental time. In the case of a much larger example with 1,000 memory points and requiring 10,000 input patterns per experiment, the time gain would be about 96%, and would correspond to about 8 months of non-stop simulation ! The proposed approach can therefore make the difference between experiments made in a few days and experiments that cannot be made in practice because they would be much too costly in development time. Of course, this does not take into account any possible optimisation of the simulation process (e.g. [28]), that may also reduce the simulation length to an acceptable value. Let us also notice that the relative time reduction is the same for two campaigns with the same number of input patterns applied per experience, but ten times more or less injected faults (third and last lines in Table 1). This is due to the fact that the injection time using the run-time reconfiguration remains very high with respect to an emulation cycle. The total time needed for the injection campaign and the absolute time saved by using emulation increase almost linearly with the number of injected faults (or the number of injection experiments). However, the relative gain depends almost only on the number of cycles per experiment.

This shows that the efficiency of RTR-based fault injection not only depends on the use of partial reconfiguration, but also on the speed of partial read back and partial reconfiguration. These points and/or the ease of control of the flip-flop set/reset signals must be carefully optimised in order to reduce the experiment time, and this of course depends on the FPGA architecture, not on the injection process. With the architecture used in this study, the reconfiguration time remains noticeable with respect to the other parameters defining the total emulation time. However, noticeable gains can be expected compared with simulations.

Number of input patterns	Number of injected faults	Simulation time (h)	Gain using emulation with RTR
1,000	100,000	67	58%
10,000	1,000,000	667	96%
10,000	10,000,000	6667	96%

Table 1. Estimated comparisons of execution times

Comparing the proposed approach with the emulation of an instrumented description, one should take into account that the instrumented description generally has an increased critical path, that may decrease the clock frequency of the prototype. However, the most significant parameter when comparing the two approaches remains the number of input patterns per experiment, because some extra injection time must be added to each experiment in the RTR-based approach. There is therefore a trade-off between the gain obtained when running the experiments with a higher clock frequency and the overhead due to the fault injection time. The RTR-based approach is thus more efficient when the number of input patterns increases. It appears, in the estimations we made on the basis of a 100 ms reconfiguration time, that the two approaches lead to similar experiment lengths when more than 1 million of input patterns are used per experiment. This limit would go down, and the RTR-based approach would be more efficient, if the speed of the run-time reconfiguration increases. In most cases, the advantage of the proposed approach with respect to the instrumented emulation would therefore not be related to the time required to run the experiments, but rather to the time potentially needed to prepare several instrumented versions when a single one cannot cope with the emulator characteristics.

5.2: Experiments with a development board

Experiments where also carried out using a Virtex board from XESS (XSV board with a XCV50 device [33]) connected to a PC via a parallel port. The fault injection application was executed in the Windows NT environment, using JBits 2.8 [34]. The frequency of the FPGA system clock was 10 MHz during the experiments. The experiments used several circuit examples, including a fuzzy logic processor specified in VHDL and already used as case study in [31].

The results obtained during the experiments with the fuzzy logic controller were quite surprising since the injection times were found much higher than expected. The measures showed an initialisation time of the FPGA (full configuration) of about 20 seconds. The injection of a fault, including the read back of the states and the modification of the switches before and after pulsing the GSR line, took about 3.5 seconds on an average instead of 100 ms or less. The other read back phases, used to capture the significant information necessary to analyse the behaviour of the circuit after the injection, were also very slow. Although the feasibility of the injection process using Jbits is validated by these experiments, it appears clearly that the time overhead obtained for each injection experiment is not acceptable.

The analysis of these measures pointed out that the reconfiguration process was considerably slowed down by the design of the board on which the FPGA is connected. The frequency of the configuration clock of the FPGA, generated on the board, was found to be only 4 kHz. Added to the serial configuration mode of the FPGA and to the speed limitations of the parallel port (less than 50 Kbps in reading mode), this explains the very low performances observed during the experiments for the read back and reconfiguration phases.

In order to take advantage of the run-time reconfiguration to perform fault injections, it is therefore fundamental to use a board designed to speed up the configuration. This includes a better interface with the host computer (for example, a PCI or USB interface instead of a parallel port). This also includes a high frequency configuration clock and a configuration done in parallel mode.

6: Conclusion

A new methodology for the injection of SEUs has been presented in this paper. In this methodology, faults are injected by modifying directly the circuit in reconfigurable hardware, avoiding any additional synthesis, mapping, place & route and bitstream generation after the bitstream of the original design has been generated. Also, no specific hardware has to be implemented to inject the faults, allowing the designer to use a smaller FPGA and resulting in a better maximal clock frequency for the prototype. The injection time overheads induced by the approach were discussed and measures made during practical experiments were reported. It has been shown that noticeable gains could be expected compared with simulation-based injection experiments, provided that the configuration of the FPGA is quick enough. This implies to optimise 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 with not only partial reconfigurability but also 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.

Developing such an efficient environment for RTR-based fault injection is a subject for further work.

References

[1] J. Abke, E. Böhl, and C. Henno. Emulation based real time testing of automotive applications. In *4th IEEE International On-Line Testing workshop*, pages 28–31, July 1998.

[2] L. Antoni, R. Leveugle, and B. Fehér. Using run-time reconfiguration for fault injection in hardware prototypes. In *IEEE International Symposium on Defect and Fault Tolerance in VLSI Systems*, pages 405–413, October 2000.

[3] L. Antoni, R. Leveugle, and B. Feher. Using run-time reconfiguration for fault injection applications. In *IEEE Instrumentation and Measurement Technology Conference*, volume 3, pages 1773–1777, May 2001.

[4] A. Benso, P.L. Civera, M. Rebaudengo, and M. Sonza Reorda. A low-cost programmable board for speeding-up fault injection in microprocessor-based systems. In *IEEE Reliability and Maintainability Symposium*, pages 171–177, January 1999.

[5] A. Benso, P. Prinetto, M. Rebaudengo, and M.S. Reorda. A fault injection environment for microprocessor-based boards. In *IEEE International Test Conference*, pages 768–773, October 1998.

[6] A. Benso, M. Rebaudengo, M.S. Reorda, and P.L. Civera. An integrated hw and sw fault injection environment for real-time systems. In *IEEE International Symposium on Defect and Fault Tolerance in VLSI Systems*, pages 117–122, November 1998.

[7] E. Böhl, W. Harter, and M. Trunzer. Real time effect testing of processor faults. In *5th IEEE International On-Line Testing workshop*, pages 39–43, July 1999.

[8] J. Boué, P. Pétillon, and Y. Crouzet. MEFISTO-L: a VHDL-based fault injection tool for the experimental assessment of fault tolerance. In *28th FTCS*, pages 168–173, June 1998.

[9] K.-T. Cheng, S.-Y. Huang, and W.-J. Dai. Fault emulation: A new methodology for fault grading. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 18(10):1487–1495, October 1999.

[10] P. Civera, L. Macchiarulo, M. Rebaudengo, M. S. Reorda, and A. Violante. Exploiting fpga-based techniques for fault injection campaigns on vlsi circuits. In *IEEE International Symposium on Defect and Fault Tolerance in VLSI Systems*, pages 250–258, October 2001.

[11] P. Civera, L. Macchiarulo, M. Rebaudengo, M. S. Reorda, and A. Violante. Exploiting fpga for accelerating fault injection experiments. In *IEEE International On-Line Testing Workshop*, pages 9–13, July 2001.

[12] P. Civera, L. Macchiarulo, M. Rebaudengo, M. S. Reorda, and A. Violante. Fpga-based fault injection for microprocessor systems. In *Asian Test Symposium*, pages 304–309, November 2001.

[13] T. A. Delong, B. W. Johnson, and J. A. Profeta. A fault injection technique for VHDL behavioral-level models. *IEEE Design and Test of Computers*, 13:24–33, Winter 1996.

[14] S. A. Guccione, D. Levi, and P. Sundararajan. JBits: Java-based interface for reconfigurable computing. In *2nd Annual MAPLD*, 1999.

[15] E. Jenn, J. Arlat, M. Rimen, J. Ohlsson, and J. Karlsson. Fault injection into VHDL models: the MEFISTO tool. In *24th International Symposium on Fault-Tolerant Computing*, pages 66–75, June 1994.

[16] Z. Kalbarczyk, R.K. Iyer, G.L. Ries, J.U. Patel, M.S. Lee, and Y. Xiao. Hierarchical simulation approach to accurate fault modeling for system dependability evaluation. *IEEE Transactions on Software Engineering*, 25:619–632, September-October 1999.

[17] Z. Kalbarczyk, G. Ries, M.S. Lee, Y. Xiao, J. Patel, and R.K. Iyer. Hierarchical approach to accurate fault modeling for system evaluation. In *IEEE International Computer Performance and Dependability Symposium*, pages 249–258, September 1998.

[18] G. A. Kanawati, N. A. Kanawati, and J. A. Abraham. FERRARI: a tool for the validation of system dependability properties. In *IEEE Symposium on Fault-Tolerant Computing*, pages 336–344, July 1992.

[19] G.A. Kanawati, N.A. Kanawati, and J.A. Abraham. Ferrari: a flexible software-based fault and error injection system. *IEEE Transactions on Computers*, 44:248–260, February 1995.

[20] J. Karlsson, U. Gunneflo, P. Liden, and J. Torin. Two fault injection techniques for test of fault handling mechanisms. In *IEEE International Test Conference*, page 140, October 1991.

[21] J. Karlsson, P. Liden, P. Dahlgren, R. Johansson, and U. Gunneflo. Using heavy-ion radiation to validate fault-handling mechanisms. *IEEE Micro*, 14:8–23, February 1994.

[22] R. Leveugle. Behavior modeling of faulty complex VLSIs: why and how? In *The Baltic Electronics Conference*, pages 191–194, October 1998.

[23] R. Leveugle. Towards modeling for dependability of complex integrated circuits. In *IEEE International On-Line Testing Workshop*, pages 194–198, July 1999.

[24] R. Leveugle. Fault injection in vhdl descriptions and emulation. In *IEEE International Symposium on Defect and Fault Tolerance in VLSI Systems*, pages 414–419, October 2000.

[25] R. Leveugle. A low-cost hardware approach to dependability validation of ips. In *IEEE International Symposium on Defect and Fault Tolerance in VLSI Systems*, pages 242–249, October 2001.

[26] T. Michel, R. Leveugle, G. Saucier, R. Doucet, and P. Chapier. Taking advantage of asics to improve dependability with very low overheads. In *European Design and Test Conference*, pages 14–18, February-March 1994.

[27] G. Miremadi and J. Torin. Evaluating processor-behavior and three error-detection mechanisms using physical fault-injection. *IEEE Transactions on Reliability*, 44:441–454, September 1995.

[28] B. Parrotta, M. Rebaudengo, M. S. Reorda, and M. Violante. New techniques for accelerating fault injection in vhdl descriptions. In *International On-Line Testing Workshop*, pages 61–66, July 2000.

[29] J. R. Samson, W. Moreno, and F. Falquez. Validating fault tolerant designs using laser fault injection. In *IEEE International Symposium on Defect and Fault Tolerance in VLSI Systems*, pages 175–183, October 1997.

[30] S. Svensson and J. Karlsson. Dependability evaluation of the THOR microprocessor using simulation-based fault injection. Technical Report 295, Chalmers University of Technology, Department of Computer Engineering, November 1997.

[31] R. Velazco, R. Leveugle, and O. Calvo. Upset-like fault injection in vhdl descriptions: A method and preliminary results. In *IEEE International Symposium on Defect and Fault Tolerance in VLSI Systems*, pages 259–267, October 2001.

[32] R. W. Wieler, Z. Zhang, and R. D. McLeod. Emulating static faults using a xilinx based emulator. In *IEEE Symposium on FPGAs for Custom Computing Machines*, pages 110–115, April 1995.

[33] XESS Corporation, 2608 Sweetgum Drive, Apex NC 27502. *XSV Board V1.0 Manual*, September 2000.

[34] Xilinx, 2100 Logic Drive. San Jose, CA 95124-3450. *JBits 2.8*, September 2001.