JOURNAL OF ELECTRONIC TESTING: Theory and Applications 19, 577–584, 2003

Accurate Analysis of Single Event Upsets in a Pipelined Microprocessor

M. REBAUDENGO, M. SONZA REORDA AND M. VIOLANTE
Politecnico di Torino, Dipartimento di Automatica e Informatica, Torino, Italy
www.cad.polito.it

Received July 21, 2002; Revised November 24, 2002

Editor: C. Metra

Abstract. Modern processors embed features such as pipelined execution units and cache memories that can not be directly controlled by programmers through the processor instruction set. As a result, software-based fault injection approaches are even less suitable for assessing the effects of SEUs in modern processors, since they are not able to evaluate the effects of SEUs affecting pipelines and caches. In this paper we report an analysis of a commercial processor core where the effects of SEUs located in the processor pipeline and cache memories are studied. The obtained results are compared with those software-based approaches provide, showing that software-based approaches may lead to significant errors during the error rate estimation. A major novelty of the paper is an extensive analysis of the effects of SEUs in the pipeline of a commercial processor core during the execution of several benchmark programs.

Keywords: dependability evaluation, single event upsets, fault injection

1. Introduction

The need for developing safety or mission critical applications with high demands in terms of computational power under low-cost constraints pushed designers to explore the possibilities offered by high-end processors. Commercial processors are bundled with features allowing them to execute a very high number of instructions per second; moreover, mostly thanks to the high number of sold units, their cost is orders of magnitude lower than that of their hardened versions. They are thus particularly attractive since they combine low cost with high performance.

In order to fruitfully exploit them in critical applications, their sensitivity to soft errors should be deeply investigated. In particular, detailed characterization of processors behavior in presence of Single Event Upsets (SEUs) is mandatory. In order to cope with the complexity of the task, which implies the ability of altering and monitoring the operations of a processor during the execution of a workload, hardware prototypes are usually exploited. Radiation testing is then normally adopted to analyze SEU effects on microelectronic devices in terms of static cross-section. Static cross-section corresponds to the sensitivity to SEUs of all the processor memory elements (registers and internal memories) and is independent on the executed program. In practice, static cross-section is often obtained by measuring the number of corrupted bits in the processor storage elements while the circuit is exposed to suitable radiation beams. Static cross-section is then combined with the figures characterizing the final environment to estimate the error rate of the final application. The obtained figure is a worst-case estimation of SEU effects because it does not take into account the impact of the executed application on the processor cross-section: for example, an application may use only a limited portion of the processor register file, and thus SEU effects on the un-used registers should be ignored during cross-section computation.

Recently, an alternative approach [8] has been proposed to overcome this limitation: the method combines software fault injection results with static cross-section figures derived from radiation experiments. In particular, static cross-section is multiplied by the ratio of injected faults producing a failure (hereinafter called *failure rate*), obtaining the application error rate.

In the outlined analysis process, failure rate computation through fault injection is crucial for obtaining accurate estimations of the application error rates. As far as simple processors are concerned, software-based fault injection [4] is commonly adopted for this task. It indeed allows injecting faults while running the application at the processor full-speed; a high number of faults can thus be analyzed in a reasonable amount of time.

When modern high-performance processors are concerned, software-based fault injection is no longer able to provide accurate estimation of the processor failure rate. These processors employ architectural solutions such as pipelined execution units, out-of-order instruction issue units and cache memories that significantly increase the number of memory elements the processors embed, and that are hidden to programmers, since no instruction is available to access them. As a result, software-based fault injection cannot be exploited to evaluate SEU effects in these un-reachable (hidden) memory elements.

On the other side, simulation-based fault injection [3, 6] can be exploited to gather more detailed information about fault effects, but its adoption is strongly limited by the enormous amount of CPU time it requires, as well as by the need for a detailed model of the processor.

The aim of this paper is twofold. To compare from an experimental point of view two alternative fault injection methods: the software-based one and the simulation-based one. The purpose of this comparison is to evaluate the error rate estimation errors stemming from the adoption of fault injection techniques that do not allow evaluating the effects of SEUs in the hidden part a modern processors (cache memory and pipelined execution unit). Moreover, the aim of this paper is to report a detailed study of the effects of SEUs in the pipeline of a modern processor. To the best of our knowledge this is the first time that the pipeline of a commercial processor was studied resorting to simulation-based fault injection for assessing the effects of SEUs. This analysis allowed identifying the most critical portion of the pipeline as well as to evaluate the effectiveness of the error detection mechanisms it embeds.

The mentioned purposes require the execution of a large set of fault injection experiments. In order to speed-up the simulation experiments, we exploited the fault injection environment described in [1], which provides a very efficient way of assessing the effects of SEUs in all the processor memory elements (including those that are not accessible via software-based fault injection).

In our analysis we considered a processor implementing the Sparc v8 architecture running some benchmark programs: for each of them we compared the error rate measured when performing two kinds of experiments:

1. We injected SEUs in the processor register file only as usually done when software-based fault injection is exploited.
2. We extended SEU fault injection to processor pipeline and cache memory elements.

Faults have not been injected in the data/code memory since we are interested in studying the effects of SEUs on the processor internal memory elements, only.

The results reported in the paper show that, when pipelined processors are considered, error rate computation cannot neglect the effects of SEUs in the processor hidden part. Moreover, they showed that the effects of faults are evenly distributed among the pipeline stages.

The paper is organized as follows. Section 2 provides the reader with background information concerning the fault model, the procedure for computing the error rate of a device and the adopted fault injection environment. Section 3 describes the analyzed processor, while Section 4 reports the obtained results. Finally, Section 5 draws some conclusions.

2. Background

This sections provides the reader with background information about the fault model we considered during our analysis, about the processor error rate estimation and about the fault injection environments we adopted.

2.1. *The Fault Model*

The fault tolerance community is increasingly concerned by the occurrence of soft errors resulting from the perturbation of storage cells caused by ionization [2]. This type of soft errors is known as Single Event Upset (SEU). A characteristic of SEUs is that they are random events and thus they may occur at unpredictable times. For example, they may corrupt the content of a processor register during the execution of an instruction.

In this paper we focused on the fault model called *upset* or *transient bit-flip*, which results in the modification of the content of a storage cell during program execution. Possible fault locations are thus internal memory cells, flip-flops, bits of user and control registers and even registers not accessible through the processor instruction set, and embedded memories such as register files and caches.

Despite its relative simplicity, the bit-flip is widely used in the fault tolerance community to model real faults, since it closely matches the real faulty behavior [5].

2.2. *Error Rate Estimation*

The error rate [8] of a device can be computed as follows:

$$\tau_{\mathrm{SEU}} = \sigma_{\mathrm{SEU}} \cdot P_{\mathrm{WA}} \qquad (1)$$

where σ_{SEU} is the SEU static cross section (in cm^2/device) of the considered device, and P_{WA} is the probability that a SEU hitting the device produces wrong answer (i.e., the device is affected by the SEU in such a way that the results it provides are different from the expected one).

The latter term can be computed by first performing a fault injection campaign, and then by computing the number of wrong answers over the number of injected faults. Conversely, σ_{SEU} is measured by performing a static test, i.e., the content of the device memory elements is continuously read during a radiation session. Static cross section thus measures the fraction of particles hitting the circuit that originates SEUs in the device memory elements.

2.3. *Fault Injection Environment*

The exploited FPGA-based fault injection environment is described in Fig. 1. A software tool instruments the model of the analyzed processor according to the instrumentation mechanisms described in [1].

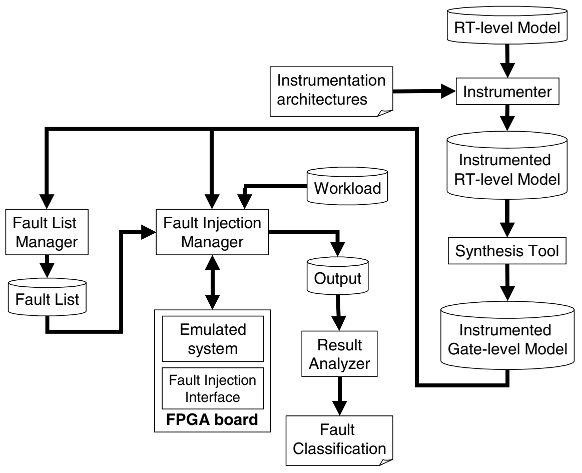

Fig. 1. The FPGA-based fault injection flow.

The obtained model is then synthesized and mapped on a FPGA device.

Two hardware platforms are used: a host computer and a FPGA board. The former acts as a master and is in charge of managing Fault Injection campaigns. The latter acts as a slave and is used to emulate the system under analysis. In particular, the FPGA-based fault injection environment exploits a FPGA board where two modules are implemented: the emulated system and the Fault Injection Interface, which allow a host computer to control the behavior of the emulated system.

A major advantage of the adopted fault injection environment is that it operates on RT-level processor descriptions, which are more widely available than gate-level ones.

The fault injection environment is composed of the following modules:

- *Fault List Manager*: This module is in charge of generating the list of faults to be injected according to the system, the input data and the possible indications of the designer (e.g., information about the most critical portions in the system).
- *Fault Injection Manager*: This module is the core of the fault injection environment, and is in charge of orchestrating the selection of a new fault, its injection in the system, and the observation of the resulting behavior.
- *Result Analyzer*: The task of this module is to analyze the output data produced by the system during each fault injection experiment, categorize faults according to their effects, and produce statistical information.

The FPGA board is driven by a host computer where the other modules and the user interface run. The fault injection manager is implemented as a hardware/software system where the software partition runs on the host, and the hardware partition is located on the FPGA board along with the emulated system.

To efficiently determine the behavior of the circuit when a fault appears, the FPGA board emulates an instrumented version of the system, which allows both the injection of each fault, and the observation of the corresponding faulty behavior.

3. Analyzed Processor

We considered the Leon core implementing the SPARC v8 architecture [http://www.gaisler.com]. The core description corresponds to about 100,000 lines of synthesizable RT-level VHDL code. The model includes 2 Kbytes of memory for implementing the instruction cache and 2 Kbytes for the data one, an integer multiplication and division units, and a 5-stage pipeline. When synthesized, this description produces a netlist of about 35,000 gates. The core has been instrumented according to the flow described in Section 2.3 and then synthesized on a Xilinx Virtex 1000E device. The obtained design amounts to 4,762 out of 12,288 logics blocks, uses 14 out of 96 block RAMs and runs at 30 MHz.

The processor pipelined integer unit is composed of the following stages:

- *Fetch*: It loads a new instruction either from the I-cache or the main memory, depending on the cache subsystem configuration.
- *Decode*: It decodes the instruction and reads the required operands. Operands may be read from the register file or from pipeline data bypass registers. This stage also takes care of address computation for *CALL* and *BRANCH* instructions.
- *Execute*: It executes arithmetical, logical and shift operations. It also computes the address for memory access instructions (e.g., *LD*) and for unconditional jump instructions (e.g., *JMPL* and *RETT*).
- *Memory*: It accesses the data cache.
- *Write*: It writes to the register file the results of any arithmetical, logical, shift or cache read instruction.

By analyzing the reports produced by the Synopsys FPGA Compiler II tool, we gathered the number of flip-flops in the circuit, which are summarized in Table 1.

Table 1. Processor memory elements.

Processor module	Number of flip-flops
Pipelined integer unit	742
Register file	4,352
Cache-memory	39,168

As the reader can observe from Table 1, the processor has a significant amount of flip-flops that cannot be directly accessed by the programmer. Indeed, the processor instruction set does not provide instructions for accessing the memory elements inside the pipeline, neither instructions for controlling the memory implementing the cache, which account for about 88% of the processor internal memory.

4. Experimental Results

This section first describes the benchmark programs we adopted during fault injection campaigns and the fault effect classification we adopted. Then, it reports a first set of experiments aiming at comparing the error rate measured by means of software-based fault injection with that coming from our FPGA-based fault injection approach. Finally, an extensive analysis of the effects of SEUs on the processor pipeline is reported.

4.1. Benchmark Programs

In our experimental analysis we considered three benchmark programs:

- *MTX*: A matrix multiplication program, where two integer matrices are multiplied. We considered two implementations of this benchmark, one working on 4×4 matrices and another one working of 10×10 matrices;
- *HS*: An implementation of the heap sort algorithm, which exploits a recursive procedure. As in the previous case, we considered two implementations of this program. One working on a set of 128 integer values, and one working on 512 values;
- *FFT*: An implementation of the fast Fourier transform algorithm adopted in digital signal processing applications.

The programs, whose characteristics are reported in Table 2, are coded in C and have been compiled

Table 2. Programs characteristics.

Program	C lines (#)	Data segment size (# byte)	Code segment size (# byte)	Execution time (# clock)
MTX 4 × 4	35	192	2,832	5,033
MTX 10 × 10	35	1,200	2,832	44,076
HS 128	60	512	8,384	13,187
HS 512	60	2,048	8,384	36,630
FFT	512	784	21,936	174,557

resorting to the GNU C/C++ gcc compiler, which is able to generate code for the Leon processor [http://www.gaisler.com].

4.2. *Fault Effects Classification*

Fault effects are classified according to the following categories:

1. *Wrong answer*: The results produced by the faulty processor are different than those produced by the fault-free processor.
2. *Effect-less*: The results produced by the faulty processor are equal to those produced by the fault-free processor.
3. *Latent*: The results produced by the faulty processor are equal to those produced by the fault-free processor, but at the end of the program execution, the content of the pipeline of the fault-free processor differs from that of the faulty one.
4. *Exception*: The injected fault is detected by the error detection mechanisms the processor embeds, which force the processor to generate an exception (e.g., illegal instruction exception or invalid address exception).
5. *Time-out*: The faulty processor is not able to produce the expected result after a given amount of time.

4.3. *Comparing Software-Based and Simulation-Based Fault Injection*

The purpose of this set of experiments is to compare a software-based fault injection approach with a simulation-based one in terms of their error rate estimation accuracy. Two fault injection campaigns have been performed, each one composed of 10,000 faults. For all of them, fault locations are randomly selected. Similarly, injection times are randomly selected among the clock cycles needed for executing the program.

- *RF*: Faults are injected in the processor register file. This fault injection campaign targets only user-accessible registers, and thus it allows gathering the same results a traditional software-based approach provides.
- *ALL*: Faults are injected in the whole processor (register file, pipeline and cache memories).

To speed-up simulations, we adopted the environment exploiting circuit emulation which is summarized in Section 2.

The experiments we performed are based on the assumption that the register file, cache memories and the processor pipeline are implemented with the same manufacturing technology. As a result, they have the same probability of being affected by charged particles hitting the circuit. For the sake of this analysis the MTX and FFT benchmarks have been adopted. Results gathered on the benchmarks during the fault injection campaigns are reported in Table 3, where the number of Wrong Answers and the corresponding error rate is reported.

The error rate was computed resorting to Eq. (1), where $\sigma_{\text{SEU}} = 2.28 \cdot 10^{-4}$ cm^2/device [3].

These results show that, when all the processor memory elements are affected by charged particles originating SEUs, the error rate significantly increases with respect to the same figure recorded when only the register file is considered. In our experiments we observed an error rate figure 73% higher when a simple application such as the MTX 4 × 4 one is considered; while for larger applications such as the FFT one, we observed an error rate figure more than 13 times higher.

These results suggest that, when complex processors are considered, the hidden memory elements (i.e., memory elements that are not accessible through the instruction set) may significantly modify the figures coming from fault injection experiments. Fault injection

Table 3. Fault injection results.

Bench.	Campaign	Wrong answer	Error rate (cm^2/device)
MTX 4 × 4	RF	151	$3.4 \cdot 10^{-6}$
	ALL	263	$5.9 \cdot 10^{-6}$
MTX 10 × 10	RF	172	$3.9 \cdot 10^{-6}$
	ALL	656	$15.0 \cdot 10^{-6}$
FFT	RF	120	$2.7 \cdot 10^{-6}$
	ALL	1,737	$39.6 \cdot 10^{-6}$

techniques providing access to hidden memory elements are thus mandatory to produce meaningful information about the effects of SEUs on modern processors. In particular, the required fault injection environments should conjugate *flexibility* in terms of fault location, to provide access to all the processor memory elements, with high *efficiency*, to allow injecting a high number of faults in reasonable amount of time.

4.4. *Analysis of Fault in the Processor Pipeline*

In this section we report the results of an analysis of SEU effects on the processor pipeline. For this purpose, we injected 10,000 randomly selected faults in each of the pipeline stages.

The results we gathered for the considered benchmarks are reported in Tables 4 to 8. All the benchmark programs have been considered for taking into account different possible scenario for the pipeline utilization. In particular, the MTX program is an example of computing intensive application that stresses the pipeline integer unit; the HS program mainly performs data transfer operations thus is intended for stressing the communication between the integer unit and the memory subsystem. Finally, FFT combines the peculiarity of the mentioned programs; moreover it is intended for further stressing the processor pipeline since it manipulates floating point values.

Table 4. Results for MTX 4 × 4.

Stage	Effect less	Latent	Wrong answer	Time out	Exception
Fetch	6,548	393	656	589	1,814
Decode	7,407	38	1,242	185	1,128
Execute	8,351	221	1,032	212	184
Memory	8,519	311	509	226	435
Write	8,432	796	229	141	402
Total	*39,257*	*1,759*	*3,668*	*1,353*	*3,963*

Table 5. Results for MTX 10 × 10.

Stage	Effect less	Latent	Wrong answer	Time out	Exception
Fetch	6,203	337	991	199	2,270
Decode	6,710	9	1,821	105	1,355
Execute	7,727	287	1,673	97	216
Memory	8,512	448	630	26	384
Write	8,325	832	253	44	546
Total	*37,477*	*1,913*	*5,368*	*471*	*4,771*

Table 6. Results for HS 128.

Stage	Effect less	Latent	Wrong answer	Time out	Exception
Fetch	7,115	84	402	508	1,891
Decode	7,535	12	864	175	1,414
Execute	8,727	171	482	342	278
Memory	8,756	0	332	456	456
Write	7,878	479	347	883	413
Total	*40,011*	*746*	*2,427*	*2,364*	*4,452*

Table 7. Results for HS 512.

Stage	Effect less	Latent	Wrong answer	Time out	Exception
Fetch	7,083	82	399	390	2,046
Decode	7,431	12	956	105	1,496
Execute	8,725	178	541	295	261
Memory	8,746	0	330	408	516
Write	7,816	486	359	883	456
Total	*39,801*	*758*	*2,585*	*2,081*	*4,775*

Table 8. Results for FFT.

Stage	Effect less	Latent	Wrong answer	Time out	Exception
Fetch	7,232	44	562	188	1,974
Decode	7,984	4	637	45	1,330
Execute	9,238	6	331	203	222
Memory	8,995	0	234	250	521
Write	8,518	440	145	282	615
Total	*41,967*	*494*	*1,909*	*968*	*4,662*

From these experimental results, several considerations arise.

The size of the program data segment does not significantly alter the effects of SEUs in the processor pipeline. As the results gathered on two versions of MTX and HS show, moving from the program with a smaller data segment to the larger one contributes to slightly decreasing the number of Effect less. As far as the remaining faults are considered (i.e., all the faults that have observable effects on the program behavior), we have that the number of Wrong answer and Latent tends to increase with the size of the data segment, while the number of Time out tends to decrease. Conversely, the number of Exception remains almost unchanged.

Significant variations in fault effects is observed when moving from one program to another one: the

Table 9. Instruction mix.

Program	Different instructions (#)
MTX	30
HS	34
FFT	53

number of faults leading to Wrong answer, Latent and Time out is indeed very different from MTX, HS and FFT.

By analyzing the number of Wrong answer for the considered benchmarks, we can see that a general trend can be forecasted. During our experiments we indeed observed that the Decode stage is the most susceptible stage to SEUs no matter which is the executed program, i.e., the SEUs injected in the Decode stage have the highest probability of originating Wrong answers.

The experiments also showed that the number of Latent decreases from the simpler MTX program to the more complex FFT one. This can be explained by considering, along with the fault injection results, the figures coming from the analysis of the assembly code of the benchmarks programs: we inspected the assembly code the C compiler produced and we counted the number of different instructions composing the program. As one can observe in Table 9, the number of different instructions composing FFT is 53, which is much higher than that of MTX (that is just 30) and HS (34). These figures suggest that FFT uses the pipeline more intensively than the other programs: during the FFT execution, most of the pipeline registers are constantly updated with meaningful information. As a result, a SEU hitting the pipeline has a lower probability of remaining latent during FFT execution.

As far as the number of fault classified as Time out is concerned, we have comparable figures for MTX and FFT, while a much higher figure for the HS benchmark. This is explained by considering that HS is based on a recursive algorithm. Faults affecting the program execution flow are thus likely to let the processor enter an end-less loop where the recursive function continuously calls itself.

Finally, the results we gathered allow studying the effectiveness of the error detection mechanism the processor embeds. The mechanism exploits the generation of an exception as soon as an error is detected: for example the processor is trying to decode an invalid instruction or it is trying to access to an invalid memory address. From the fault injection campaign we performed, we observe that the number of faults classified as Exception seems to be independent by the application the program executes. For all the considered benchmarks we measured almost the same number of faults producing exceptions. Moreover, if we consider the distribution of faults classified as Exception within the five pipeline stages, we observed that most of them are located in the Fetch and Decode stages: the number of exceptions observed for these two stages is indeed about 7 times higher than that measured for the other ones. This let us conclude that the error detection mechanism works independently from the program the processor is running, but it deals with faults originated in the Fetch and Decode stages, only.

5. Conclusion

This paper analyzed from an experimental point of view the effects of SEUs inside a modern processor embedding advanced architectural features such as a five-stages pipeline and data/instruction cache memory.

From the one hand, the results we obtained showed how simulation-based fault injection techniques prevails over the software-based one, as far as the analysis of processors embedding hidden memory elements is concerned. For this kind of processors the accuracy simulation-based techniques provide is much higher (up to 13 times higher) than software-based one. Moreover, the results we gathered underlined the need for efficient techniques for speeding-up simulation-based fault injection campaigns.

From the other hand, an extensive analysis of the SEU effects in the pipeline of the Leon processor (implementing the Sparc v8 architecture) allowed us to better understand the dependency of faulty behavior from the executed program. In particular, the experiments allowed to assess the effectiveness of the error detection mechanism the Leon core already embed, which seems to be very effective in dealing with faults hitting the Fetch and Decode stages, only.

Acknowledgment

This work has been partially supported by the Italian Ministry for University through the Center for Multimedia Radio Communications (CERCOM).

References

1. P.L. Civera, L. Macchiarulo, M. Rebaudengo, M. Sonza Reorda, and M. Violante, "FPGA-Based Fault Injection for Microprocessor Systems," in *Proc. IEEE Asian Test Symposium*, 2001, pp. 304–309.
2. E. Dupont, M. Nicolaidis, and P. Rohr, "Embedded Robustness IPs for Transient-Error-Free ICs," *IEEE Design and Test of Computers*, vol. 19, no. 3, pp. 560–570, 2002.
3. J. Gracia, J.C. Baraza, D. Gil, and P.J. Gil, "A Study of the Experimental Validation of Fault-Tolerant Systems Using Different VHDL-Based Fault Injection Techniques," in *Proc. IEEE International On-Line Testing Workshop*, 2000, pp. 73–79.
4. R.K. Iyer and D. Tang, "Experimental Analysis of Computer System Dependability," Chapter 5 of *Fault-Tolerant Computer System Design*, D.K. Pradhan (Ed.), Prentice Hall, 1996.
5. M. Nicolaidis, "Time Redundancy Based Soft-Error Tolerance to Rescue Nanometer Technologies," in *Proc. IEEE 17th VLSI Test Symposium*, April 1999, pp. 86–94.
6. B. Parrotta, M. Rebaudengo, M. Sonza Reorda, and M. Violante, "New Techniques for Accelerating Fault Injection in VHDL Descriptions," in *Proc. IEEE International On-Line Test Workshop*, 2000, pp. 61–66.
7. M. Rebaudengo, M. Sonza Reorda, and M. Violante, "An Accurate Analysis of the Effects of Soft Errors in the Instruction and Data Caches of a Pipelined Microprocessor," in *DATE2003: Design, Automation and Test in Europe*, 2003, pp. 602–607.
8. R. Velazco, S. Rezgui, and R. Ecoffet, "Predicting Error Rate for Microprocessor-Based Digital Architectures Through C.E.U. (Code Emulating Upsets) Injection," *IEEE Transactions on Nuclear Science*, vol. 47, no. 6, pp. 2405–2411, 2000.

Maurizio Rebaudengo received the M.S. degree in Electronics in 1991, and the Ph.D. degree in Computer Science in 1995, both from the Politecnico di Torino, Torino, Italy. In 1997 he became an Assistant Professor at the Department of Computer Science and Automation of the same Institution. His research interests include diagnosis and test of embedded memories, and dependability analysis of computer-based systems.

Matteo Sonza Reorda took his M.S. and Ph.D. degrees in Electronics (1986) and Computer Science (1990) from Politecnico di Torino, Italy. Since 1990 he works with the Department of Computer Science and Automation of the same Institution, where he is now a Full Professor. His interests are in the areas of Testing of Electronic Systems: in this area he published more than 150 papers on international journals and conference proceedings. He currently serves in the Program Committee of many events. He was the Program Chair of the IEEE On-line Testing Symposium in 2003.

Massimo Violante received the M.S. and Ph.D. degrees in Computer Engineering from Politecnico di Torino, Italy, in 1996 and 2001, respectively. Since 1996 he has been with the Department of Computer Science and Automation of Politecnico di Torino, where he is currently an Assistant Professor. His research interests include test and dependability analysis of digital systems.