New Techniques for Speeding-up Fault-injection Campaigns

L. Berrojo, I. González
Alcatel Espacio, S.A.
Madrid, Spain
http://www.alcatel/espacio

F. Corno, M. Sonza Reorda, G. Squillero
Politecnico di Torino
Dipartimento di Automatica e Informatica
Torino, Italy
http://www.cad.polito.it

L. Entrena, C. Lopez
Universitad Carlos III
Area de Tecnologia Electronica
Madrid, Spain
http://www.uc3m.es

Abstract*

Fault-tolerant circuits are currently required in several major application sectors, and a new generation of CAD tools is required to automate the insertion and validation of fault-tolerant mechanisms. This paper outlines the characteristics of a new fault-injection platform and its evaluation in a real industrial environment. It also details techniques devised and implemented within the platform to speed-up fault-injection campaigns. Experimental results are provided, showing the effects of the different techniques, and demonstrating that they are able to reduce the total time required by fault-injection campaigns by at least one order of magnitude.

1 Introduction

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. Finally, the adoption of new technologies for the implementation of electronic devices asks for effective techniques for making them able to guarantee a sufficient level of reliability [1].

In this framework, evaluation of the dependability of designed systems is a key point, and fault injection emerged as a viable solution [2] for the qualification plan of a design. When assessing the reliability of in-house designed ASICs or FPGAs, simulated fault injection [3], [4] is normally preferred to other approaches, such as those based on hardware fault injection [5], [6]. This is due to several reasons:

- First, simulated fault injection provides the maximum flexibility in terms of supported fault models.
- Second, it allows performing reliability assessment at different stages in the design process, well before than a prototype is available.
- Finally, simulated fault injection can normally be rather easily integrated into already existing design flows.

As a major drawback, simulated fault injection can be unacceptably slow, 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.

Several techniques have been proposed in the past to efficiently implement simulation-based fault-injection campaigns for transient faults:

- A first approach [3], [4] is based on modifying the system description, so that faults can be injected where and when desired, and their effects observed, both inside and on the outputs of the system. 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, which support the injection and observation features. This approach normally provides the best performance, but it can only be followed when the code of the simulation tools is available and easily

* This work has been partially supported by the European Community through *IST* project *AMATISTA* and by Ministero dell'Istruzione, dell'Univerità e della Ricerca through project *ISIDE*.

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.

- A third approach [7] 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 [8].

In this paper we outline the fault-injection platform that has been developed and is currently being evaluated in a real industrial environment, and describe a set of techniques devised and implemented within the platform to speed-up fault-injection campaigns. This research is performed in the context of the European IST project AMATISTA, whose main target is indeed the development of a set of tools for the design of fault tolerant circuits at RT-level.

The fault-injection platform is mainly used for assessing the correctness and effectiveness of the fault tolerance mechanisms implemented within the ASIC and FPGA designs developed for space applications. The platform works on RT-level VHDL descriptions which are then synthesized, and is based on commercial tools for VHDL parsing and simulation. Simulation-based fault injection is adopted, and prototypical tools have been developed for automatically generating the script commands interacting with the simulator.

In this paper, only single bit flip faults on memory element will be considered. The motivation is that, in synchronous designs with moderately slow clocks, transient faults are usually relevant for memory elements only. Moreover, when designs are described using well-defined synthesizable description styles, memory elements may be deterministically recognized in the RT-level source. Since, gate-level optimization algorithms usually preserve memory elements, gate-level bit flips on such memory elements can be modeled in a nearly exact way at the RT-level.

In the AMATISTA project, speeding-up RT-level fault-injection campaigns are obtained by mainly following two avenues of attack: first, clever techniques have been devised to generate and collapse the list of faults to be injected. Secondly, several optimization mechanisms have been defined and successfully evaluated to reduce the time required to simulate each fault

A prototype of the whole fault-injection platform has been implemented and it has been evaluated on a real benchmark circuit. Results are provided, showing the effects of the different techniques, and demonstrating that they are able to reduce the total time required by a fault-injection campaign by at least one order of magnitude.

The paper is organized as follows: Section 2 describes the whole fault-injection campaign, detailing fault-injection schema and fault-collapsing strategies. Section 3 reports some experimental results on an industrial design. Section 4 concludes the paper.

2 RT-Level Fault-injection Campaign

Single bit flip faults are generally termed *single error upset* (SEU). Let us denote the set of all faults with Ψ, and SEU number i with S_i. $S_i = (FF_{L_i}, T_i^A)$, where L_i is index of the fault location into flip-flop list (FF_j), i.e., the memory element that changes its value; and T_i^A is the fault activation time, i.e., the time instant when the fault location flips its value. Defining T_{sim} as the workload length, $\forall S_i \in \Psi : T_i^A \le T_{sim}$.

Function C(t) represents the state of the fault-free design at test-bench instant t. The state takes into account all values produced on output ports and all values stored into memory elements: C(t) = {PO(t), FF(t)}. $C^f_i(t)$ represents the same state under the effect of S_i. Clearly,

$$\forall t \in [0, T_i^A[: \mathrm{C}(t) = \mathrm{C}_i^{\mathrm{f}}(t)$$

since before T_i^A, $C^f_i(t)$ is not affected by S_i. The fault-free simulation is usually termed *golden run*.

It should be noted that the design is assumed to be synchronous sequential with, possibly, many clocks. By defining the quantum τ as the greatest common divider between all clocks periods, all significant time instants can be expressed in the form $a\tau$, with $a \in \mathbf{N}$. Functions C($\cdot$) and $C^f_i(\cdot)$ are discrete and the golden run is a finite list of values (C(τ), C(2τ), …, C($T_{sim}\tau$)), where $T_{sim}\tau$ is the length of the test bench. To ease formulas, in the following $\tau = 1$.

The goal of the fault-injection campaign is to grade possible faults, by partitioning the set Ψ of all faults into four different sets:

- Φ (*Failure*): the set of all SEUs that, during the test bench, produce a difference on an output port of the design.
- Σ (*Silent*): the set of all SEUs that, compared to the golden run, never produce differences on output ports and, at the end of the test bench, left no differences in memory elements.
- Λ (*Latent*): the set of all SEUs that, compared to the golden run, never produce differences on output ports, but, at the end of the test bench, cause at least a memory element to differ.
- E (*Error*): the set of all SEUs that cause an error in VHDL simulation.

At the end of the fault-injection campaign, each SEU is classified in exactly one set.

Faults belonging to the E (error) set represent a typical problem of high-level fault simulation, and they have no correspondent classification in gate-level campaigns. Simulation errors are caused, for instance, whenever a fault sets a signal to a value out of its declaration range. Once such an error has occurred, simulation may be halted. Indeed, several commercial VHDL simulators do halt simulation automatically in presence of such errors.

Failure set (Φ) deserves no special comments. As soon differences propagate to an output port, simulation may be halted and the fault classified as failure.

A fault in the latent set (Λ) may be classified whenever as soon as all fault effects disappear (e.g., at time instant t_r). Since no interference may exist between different faults, differences may be *generated* only at fault activation time, and then are *propagated*. More formally:

$$t_r \in]T_i^A, T_{SIM}] \wedge \mathrm{C}(t_r) = \mathrm{C}_i^{\mathrm{f}}(t_r)$$
$$\Rightarrow \forall t \in [t_r, T_{SIM}] : \mathrm{C}(t) = \mathrm{C}_i^{\mathrm{f}}(t)$$

Thus, in three cases (*error*, *failure*, *silent*) SEUs are categorized before the end of the test bench, while in one (*latent*), the fault-injection experiment is required to reach the end of the test bench. The possibility to classify a SEU before the end of the test bench gives the opportunity to optimize the process by stopping the simulation.

Next section details the fault-injection algorithm, while fault collapsing techniques are analyzed in 2.2, 2.3 and 2.4. Fault-collapsing methodologies that can be applied only during the fault-injection experiment are called *dynamic*, while methodologies that can be applied before it are termed *static*. Static techniques are again classified in *workload-dependent* and *workload-independent*, whereas they require the analysis of the test bench or not.

2.1 Fault Injection

The basic idea of the fault-injection algorithm is to fully simulate the fault-free design, storing the golden run. Then simulate each fault sequentially by loading the state of the design just before fault activation time, injecting the SEU and eventually categorizing its effects. Since the design is sequential, it is sufficient to load values into memory elements FF(t) and set input stimula PI(t).

Fault-injection schema is shown in Figure 1; however, a few additional performance considerations still need to be made.

```
Simulate fault-free design and store checkpoints CP = (
C(tg1), C(tg2), ..., C(tgNg) )
For each Si {
    Ei = ∅                  // Empty equivalent set
    s = max_{1≤j≤Ng, t_j^g ≤ T_i^A − 1} (j)   // Select the nearest state in GR
    Cfi(tsg) ← C(tsg)       // Load nearest state CP list
    Simulate design until t = TiA-1
    Inject Si and calculate Cfi(TiA)
    inc = 1                 // Initialize increment step
    Tend = TiA              // Initialize interval
    while(Si is not categorized) {
        Tstart = Tend
        Tend = Tstart + inc
        Simulate design calculating Cfi(t),
                                    t ∈ ]Tstart, Tend]
        Insert all faults Sj dynamically equivalent to Si
                                    in set Ei
        Try to categorize Si:
            if exists a Sj in set Ei already categorized
            or comparing Cfi(t) to C(t), t ∈ ]Tstart, Tend]
        inc = inc * 2
    }
    Categorize all faults Sj in Ei like Si
}
```

Figure 1: *Fault-Injection Schema*

Firstly, saving circuit states is a time- and space-consuming task. It is necessarily to trade-off the ability to resume simulation from *any* possible T_i^A with the amount of disk space required to save all these states. Thus, a list of *checkpoints* CP is defined as a list of N_g equally distributed simulation states $C(t_{N_g}^g)$. In order to inject fault S_i, the design is first brought to the nearest CP state preceding T_i^A-1 by loading data from the checkpoint list. Then, the fault-free design is simulated until T_i^A-1. The number N_g of checkpoints and the interval between them must be carefully regulated.

Secondly, also checking the state of a simulation, seeking differences from the golden run is a resource-consuming task. In the current implementation of the fault injector, it is significantly optimized and exploits simulator facilities, however, it still deserves special attention. It has been experimentally observed that most of the SEUs either can be classified in the first clock cycles after injection, or after a relatively long time. Thus, the simulation is run for exponentially increasing amounts of time: one clock cycle after the activation of the SEU the faulty circuit signal traces are compared against the good ones; next comparison is performed after two clock cycles, then four, and so on.

A set E_i of faults equivalent to S_i is dynamically built during simulation. The goal is to reduce the number of SEUs injected either by categorizing some faults before simulation, or stopping the current simulation as soon as the current SEU S_i is discovered equivalent to an already-simulated one. This optimization will be detailed later.

2.2 Workload Independent Fault Collapsing

This Sub-section illustrates static fault-collapsing techniques that are uniquely based on the analysis of the design.

The study of the topology of the circuit helps determining the category of a fault. First, a scheme of the circuit where only two types of elements appear: sequential and combinational blocks (e.g., Figure 2).

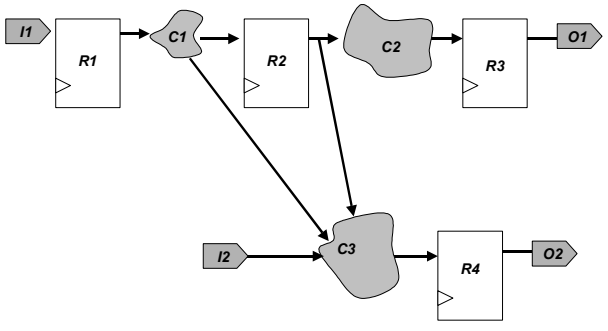

Figure 2: *Design Topology*

In this scheme it is possible to find the following cases:

- Primary input directly going to a flip-flop
- Primary input going to a flip-flop through a combinational part
- Direct communication between two flip-flops
- Communication between two flip-flops through a combinational part
- Feedback in a flip-flop
- Direct communication from a flip-flop to a primary output of the circuit
- Communication from a flip-flop to a primary output through a combinational part

With the analysis of these elements it is possible to find *dominances* and *static equivalences* with respect to the faults to be injected. Dominant fault are those fault whose effect is the same as the effect for other faults and their simulation causes the same changes as the others but not in the contrary. On the other hand, equivalent faults are those faults whose effects are the same after a period of time are considered equivalent if the effects caused by any of them are not sub-set of the effects of the others. Dominances and static equivalencies may reduce the size of the fault list, and speed-up the simulation process improving dynamic equivalencies, described later.

Direct flip-flops to the outputs are very common in current designs, aero-spatial industry imposes this condition to all its designs in order to ensure Fault Tolerance in their equipments as indicated in the *ESA Guidelines*. All flip-flops whose value is connected directly with the outputs of the circuit will have automatically all of their faults marked as failure. Moreover any flip-flop FF_i whose output is connected directly and exclusively to another flip-flop FF_j have its faults belonging to the same category as faults of the connected flip-flop FF_j.

Further optimizations are possible, considering that all SEUs affecting registers with a given bit width can be categorised by analysing faults in a single flip-flop of the register, if the same operations and transformations are affecting all the bits. In addition, whenever the output of an internal counter is the overflow or a related function, each fault affecting it can be categorized by analyzing an equivalent SEU in the last flip-flop of the counter, although with a different activation time.

Experimental evidence suggests that static test-bench dependent fault collapsing enables pruning about 10% of the total number of faults.

2.3 Workload Dependent Fault Collapsing

This Sub-section illustrates static fault-collapsing techniques that are based on the simulation of the workload, or test bench.

The fault-free design is simulated, and all read and write operations on memory elements are tracked. Operations are tracked with bit granularity (the same granularity as SEUs). That is, each single read and write operation on each single bit is logged. Figure 3 show an example of read and write operations on bit 3 of a signal named `sig`.

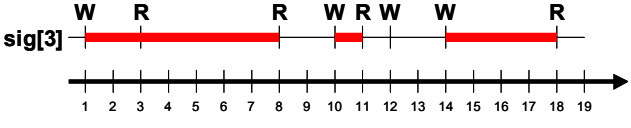

Figure 3: Golden-Run Fault Collapsing

Starting with the log, all possible SEUs are collapsed using the following rules:

- All SEUs between an operation (either read or write) and a write operation are marked as *silent*. Fault injection is useless, because their effect will be masked before any possible propagation. In the example, a SEU at $T^4 = 9$ is silent.

- All SEUs between an operation (either read or write) and the subsequent read operation are equivalent. In the example, fault on `sig[3]` with $T^A = 1$ and $T^A = 2$ are equivalent, while the SEU at $T^A = 4$ is equivalent to the one at $T^A = 7$.

Experimental evidence suggests that workload dependent fault collapsing may enable pruning up to 80% of the total number of faults.

2.4 Dynamic Fault Collapsing

This Sub-section illustrates fault-collapsing techniques that may be activated during the fault-injection campaign. As anticipated before, this technique is exploited to discover equivalencies between faults.

During the simulation of SEU S_i, if at time $t_e > T_i^A$ differences between $C(t_e)$ and $C^f_i(t_e)$ are limited to the value of exactly one memory element FF_e, then S_i is equivalent to a bit-flip S_e on that memory element with $T_e^A = t_e$. I.e.,

$$S_j \approx S_e \text{ with } S_e = (FF_e, t_e)$$

Indeed, S_e may not be explicitly listed in the fault list, because static fault collapsing marked it equivalent to a different SEU S_f. In this case, for the transitive property, S_i will be marked equivalent to S_f

$$S_j \approx S_e \wedge S_e \approx S_f \Rightarrow S_j \approx S_f$$

As a result, during simulation the set E_i of SEUs equivalent to S_i is dynamically built. Whenever the fault injector is able to categorize S_i, all faults in E_i get the same classification.

It must be also noted that the newly discovered equivalent fault S_e may be already classified, even if $T_e^A > T_i^A$. First of all, there is no reason to presume that faults are injected in the same time order of their activation time (indeed, several optimizations are currently under study to optimize the order of injections). Moreover, fault S_e may be already classified because it has been found equivalent to a fault S_k with $T_e^A > T_i^A > T_k^A$. In this eventuality, fault S_i and all elements of E_i take the same classification as S_e.

Experimental evidence suggests that, exploiting dynamic fault collapsing, about 5% of a statically-collapsed fault list may not be injected. Using a complete (not collapsed) list of SEUs, about 2 faults out of 3 may usually be classified without simulation.

3 Experimental Results

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*™ parser, fault-list generation takes advantage of Synopsis VHDL Simulator, while the fault injector is currently based on *Modelsim*™ by Model Technology. Simulation states are saved using the *checkpoint* command, and subsequently loaded exploiting the *restore* option of the simulator. The faulty circuit and the golden run are compared taking advantage of the *waveform comparison* facilities built in the simulator.

The new version of the fault injector will be closely fastened to FTL Systems' *Auriga*™ simulation system. This would lead to a closer integration, better performance and allow additional optimizations.

The available prototype was used to assess the reliability of a partially hardened version of the *Solar Array Drive Electronics* (SADE). SADE is a module developed by *Alcatel Espacio* that will be hosted on satellites and it is dedicated to rotate two solar array drives so as to get the maximum energy from the solar cell panels.

On the one hand, SADE gets control commands from and transmits telemetries to special modules via a MACS bus. On the other hand, SADE controls the two SADM (*Solar Array Drive Modules*). These two functions, interfaces and SADM control, are included in an FPGA. SADE also includes an AC/DC and telemetry module and two motor drives modules, one for each SADM. The selected foundry for this design is Actel. The technology is the 0.8 micron CMOS Radiation Hardened FPGA Family. The device shall be an RT14100A-CQ256E FPGA.

SADE contains 480 memory elements, and the typical analyzed workload consists of $T_{sim} = 131{,}026$ simulation cycles. The complete fault list Ψ contains 62,892,480 SEUs, however, in the preliminary experiments only the *motor operator* block has been considered. This block includes 95 memory elements and, since the full workload was considered, adopted complete fault list counts 12,709,522 SEUs. Since there are two motor blocks in the SADE design, current experiments take into account 40% of total faults. Examined blocks do not contain any peculiar characteristic. Table 1 summarizes the result of the fault collapsing.

	#	%
Total number of SEU	12,709,522	100.00
Pruned by workload independent collapsing	1,379,720	10.86
Pruned by workload dependent collapsing	9,448,798	74.34
SEU that need to be injected	1,618,952	12.74

Table 1: Fault Collapsing Experiment

At the end of the fault-collapsing phase, the fault list has been collapsed to about 13% of its initial size. Then, during fault-injection of a sampled fault list, dynamic

equivalencies allow to avoid simulation of a further 4.73% of the sampled SEUs. It should be remarked that dynamic fault collapsing does not introduce any significant overhead in the simulation.

Experiments where run on a SPARC ULTRA Workstation with 256MB of RAM. The CPU time required for running the preliminary golden run simulation of SADE was about 80 seconds. This time includes generating and saving all required checkpoints. For the sake of comparison, the CPU time required running a full simulation of the test bench is about 6 seconds, but this time accounts for no comparisons of any kind: the simulation is run without observing nor storing any information.

The CPU time required for simulating a single fault was about 5 seconds. It is only slightly smaller than running complete simulation, but for each fault the fault injector loads a state from the golden run, and runs the simulation checking waveform and massively interacting with the simulator.

4 Conclusions

This paper described a set of techniques for speeding-up fault-injection campaigns on fault tolerant circuits at RT-level.

Experimental results illustrated the effectiveness of the proposed approach in term of fault list collapsing and fault injection mechanism. The evaluation in an industrial environment showed that a SEU can be injected and categorized in a reasonable amount of time on a workstation, and only a small fraction of possible SEUs needs to be explicitly injected.

In more general terms, through integration into commercial design flows, the *AMATISTA* project hopes to increase the usage of fault tolerant technology in application sectors where the time and additional people previously required to manually implement approaches has not resulted in wide-spread use of fault tolerance.

Industrial partners foresee that the enhancement of the design flow would allow a reduction between 25% and 35% in terms of overall design time. Improvements will include reduction of simulation times, reduction of re-cycling in the design flow, earlier detection of errors, and minimization of several design effort.

Experimental results show the efficacy of all fault-collapsing techniques: workload independent, workload dependent and dynamic. Dramatic reductions in the number of SEUs are essential to keep fault-injection campaigns feasible.

Deeper topological analyses of the circuits are currently under study. A *fault dictionary* for classifying SEU will be created analyzing the design starting from its outputs to its inputs, through internal registers. This dictionary will be subsequently exploited during fault collapsing or fault injection.

Other experimental results gathered on different circuits from different industries confirm the general validity of our claims.

5 References

[1] M. Nikoladis, Time Redundancy Based Soft-Error Tolerance to Rescue Nanometer Technologies, IEEE 17th VLSI Test Symposium, April 1999, pp. 86-94

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

[3] E. Jenn, J. Arlat, M. Rimen, J. Ohlsson, J. Karlsson, *Fault Injection into VHDL Models: the MEFISTO Tool*, Proc. FTCS-24, 1994, pp. 66-75

[4] T.A. Delong, B.W. Johnson, J.A. Profeta III, *A Fault Injection Technique for VHDL Behavioral-Level Models*, IEEE Design & Test of Computers, Winter 96 (Vol 13, No. 4)

[5] J. Arlat, M. Aguera, L. Amat, Y. Crouzet, J.C. Fabre, J.-C. Laprie, E. Martins, D. Powell, *Fault Injection for Dependability Validation: A Methodology and some Applications*, IEEE Transactions on Software Engineering, Vol. 16, No. 2, February 1990

[6] J. Karlsson, P. Liden, P. Dahlgren, R. Johansson, U. Gunneflo, *Using Heavy-Ion Radiation to Validate Fault-Handling Mechanisms*, IEEE Micro, Vol. 14, No. 1, pp. 8-32, 1994

[7] D. Gil, R. Martinez, J. V. Busquets, J. C. Baraza, P. J. Gil, *Fault Injection into VHDL Models: Experimental Validation of a Fault Tolerant Microcomputer System*, Dependable Computing EDCC-3, September 1999, pp. 191-208

[8] B. Parrotta, M. Rebaudengo, M. Sonza Reorda, M. Violante, "New Techniques for Accelerating Fault Injection in VHDL descriptions", *IEEE International On-Line Test Workshop*, July 2000, pp. 61-66