Test Set Generation almost for Free using a Run-Time FPGA Reconfiguration Technique

Alexandra Kourfali
Department of Electronics and Information Systems
Ghent University
Sint-Pietersnieuwstraat 41, B-9000 Ghent, Belgium
Email: alexandra.kourfali@ugent.be

Dirk Stroobandt
Department of Electronics and Information Systems
Ghent University
Sint-Pietersnieuwstraat 41, B-9000 Ghent, Belgium
Email: dirk.stroobandt@ugent.be

Abstract—The most important step in the final testing of fabricated ASICs or the functional testing of ASIC and FPGA designs is the generation of a complete test set that is able to find the possible errors in the design. Automatic Test Pattern Generation (ATPG) is often done by fault simulation which is very time-consuming. Speed-ups in this process can be achieved by emulating the design on an FPGA and using the actual speed of the hardware implementation to run proposed tests. However, faults then have to be actually built in into the design, which induces area overhead as (part of) the design has to be duplicated to introduce both a faulty and a correct design. The area overhead can be mitigated by run-time reconfiguring the design, at the expense of large reconfiguration time overheads. In this paper, we leverage the parameterised reconfiguration of FPGAs to create an efficient Automatic Test Pattern Generator with very low overhead in both area and time. Experimental results demonstrate the practicality of the new technique as, compared to conventional tools, we obtain speedups of up to 3 orders of magnitude, 8X area reduction, and no increase in critical path delay.

Keywords-Test set generation; Fault Emulation; FPGA Reconfiguration; Parameterised Configurations

I. INTRODUCTION

Ensuring a design's functional correctness is very crucial in current technologies. Integrated circuits are becoming more and more susceptible to errors as manufacturing problems are easier to occur, due to the constant decrease of CMOS feature sizes. One of the biggest challenges for today's ASIC design teams is the complexity of testing. A design's functional correctness is tested, under possible technological deviations.

Meanwhile, Field Programmable Gate Arrays (FPGAs) have become widely used, instead of ASICs, for a digital circuit's implementation. FPGAs are ICs that are produced in such a way that they can be reprogrammed or reconfigured multiple times after being manufactured. By configuring an FPGA, the user can change the functionality of the device and make it behave like any digital system. With the use of FPGAs the time-to-market is reduced because of their flexibility. Moreover, FPGAs are often used as prototypes (first actual chip) or simply as an implementation which can be rapidly tested, while the design can still be changed before the final ASIC implementation (this is called FPGA emulation).

In testing a design (after fabrication or FPGA implementation), the crucial point is having the right test set which should be small enough but also have enough tests to cover most of the possible errors. Within automatic test pattern generation fault simulation is important. The main reason to use fault simulation is that one cannot test all input combinations so one has to drastically limit the total number of input combinations. In order to be sure that this limited set covers all (or most) possible errors, we need to guess what errors may occur and therefore we need fault simulation. However, due to necessary sequential computations, the time needed for fault simulation is prohibitively large. The FPGA's reconfigurability can be used to detect if a design will function properly. This procedure is known as (FPGA-based) fault emulation and has proven more efficient than software-based methods. Therefore, reconfiguration itself can be used to expedite the ATPG process, with often an impact on area and time. Reducing these is the main focus of this paper.

Traditionally, the functionality of an FPGA circuit is represented by the bitstream that specifies the FPGA's internal logic and routing configuration. We produce an intermediate bitstream that represents all the bits as Boolean functions of parameters that describe the behaviour of faults. The proposed methodology allows the FPGA to be reconfigured during runtime very efficiently based only on the evaluation of these Boolean functions in order to reconfigure and create a test set. The advantage of this technique is that there is minimal area and runtime overhead during the creation of the test set and it needs no iterative executions of the computationally intensive design re-synthesis, mapping, placement and routing.

At a high level, our approach works as follows. In order to create a new test set generator, the tool works in two stages: the offline stage and the online stage. The first stage creates a new design that has a selected fault model injected in the initial design. Then, this fault model is mapped to abstract logic and routing resources of the FPGA, in a way

that doesn't occupy extra space. During the online stage, the FPGA is reconfigured multiple times to apply each time a new fault. This dynamic mapping in the abstract logic and routing resources, boosts reconfigurability and thus, the ATPG. As compared with other fault emulation methods, we achieve almost no area increase. We can control different faults with the use of the dynamic specialisation of the FPGA's logic and routing resources. Finally, with the use of this technique we observe that the critical path delay also stays the same as in the initial design, before the fault injection. Therefore, our ATPG method has no impact on the original design and requires almost no additional overhead.

II. PREREQUISITES

A. Automatic Test Pattern Generation

Most ATPG methods rely on fault injection. Fault injection can be realised either by fault simulation or fault emulation. Fault simulation is based on the insertion of a fault model into a Circuit Under Test (CUT), and the simulation of the design under faulty conditions in order to find a set of input values that can produce an error at one of the outputs for that particular fault. While software-based simulations offer results of good quality they are often very impractical due to their limited speed and memory consuming calculations. Furthermore, as the complexity of integrated circuits continues to increase, consistent with Moore's Law, fault simulation becomes infeasible. In order to overcome these limitations, circuit designers have turned to FPGAs for the emulation of their complete systems. In FPGA emulation, the faults are built in the hardware and tests can be applied at actual hardware speeds, thus gaining significantly in test set generation time. However, as all faults need to be emulated in hardware, this induces a very large area cost, unless the same hardware can be reused to emulate different faults. This can be done in FPGAs, as explained next.

B. FPGA architecture

An FPGA is a two-dimensional array of programmable logic blocks with a routing network to connect the blocks. SRAM memory cells can be programmed to describe the truth tables of the logic blocks' function, which are implemented by the look-up tables (LUTs). For connecting the logic blocks within the routing structure, again SRAM cells drive the inputs of multiplexers. All these SRAM cells together define the functionality of the FPGA and the bits that are stored in these cells together form the FPGA's bitstream. The FPGA's bitstream is used to program the FPGA by shifting into the SRAMs a sequence of 0s and 1s. Therefore both logic and routing infrastructure of the FPGA are completely determined by the values of the bitstream.

To reduce the area needed to emulate all circuit faults in the FPGA, the FPGA can be reconfigured multiple times in order to create a test set. Every time tests for a new fault
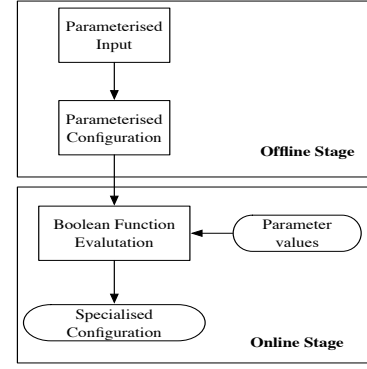

Figure 1. The two-stage tool flow of parameterized reconfiguration

need to be found, a time consuming re-synthesis of the logic circuit (with new fault present) is required, which is often prohibitive.

C. FPGA - based ATPG

We present the most common techniques for FPGA-based fault emulation that try to avoid the large area or large computation times needed for emulation based ATPG. There are two basic methodologies, in order to create an FPGA-based fault injection tool.

1) *Reconfiguration based fault emulation*, that modifies directly the configuration bitstream of the design to emulate faults. Fault emulation platforms can be used such as JBits and [4], [5], but they cannot support state-of-the-art FPGAs yet [3]. Direct bitstream manipulation has also been proposed to inject faults in LUTs [11], however not all faults can be covered with this technique. Also, changes to the HDL design have been proposed, but this again requires recompilation for every new fault injection (or one needs a lot of memory to store all possible bitstreams for every fault).

2) *Circuit instrumentation based fault emulation* techniques modify the structural descriptions of a circuit by adding extra hardware for fault injection. This avoids the time-consuming recompilations for generating a new bitstream every time a new fault needs to be injected [9], [10]. Injecting multiple faults to avoid unnecessary recompilations has also been proposed [8]. However, the size of the new design is directly proportional to the hardware complexity of the injector making the technique again infeasible for large designs.

Both circuit instrumentation-based and reconfiguration based approaches introduce specialisation overhead, which is the extra resources and the extra time needed in order to generate a test set for a specific design. Both techniques, have their limitations, regarding their use in ATPG,
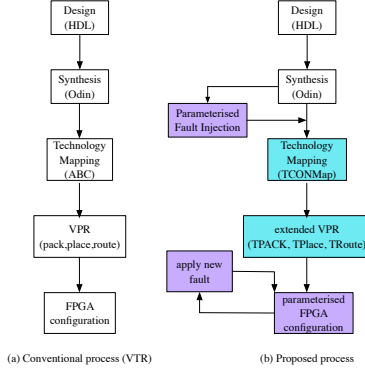
Figure 2. The conventional VTR tool flow and the VTR-TCON extended version for fault emulation

as was mentioned in Section II-B. Similar problems with specialisation overhead are found in run-time reconfiguration approaches in FPGAs outside of the field of ATPG. The authors in [1] address the problems of specialisation overhead by using the *Parameterised Configuration technique* (PConf), that dynamically specialises the logic and routing infrastructure of the FPGA, creating an implementation that can be more cost efficient in terms of area and time. The process is described in figure 1, where we can see the two-stage tool flow starts from an HDL (Hardware Design Language) description in which some less frequently varying signals, called *parameters*, are selected and annotated. These parameter signals will be assumed fixed for a certain time so that the implementation can be optimized for the fixed value. Then, when the parameter does change, the FPGA is reconfigured to a new optimized implementation for a different parameter value. This results in a bitstream that now consists of multivalued boolean functions instead of just bits. Then, at runtime these functions are rapidly evaluated to result in a regular bitstream. Parameterised Configurations enable the logic and routing infrastructure to be dynamically reconfigured with very low overhead. We therefore aim to eliminate the specialisation overhead during ATPG, with the use of the PConf approach.

III. PARAMETERISED CONFIGURATIONS BASED ATPG

We propose to add a virtual multiplexer network that inserts the right faults at the right place, instead of creating a separate circuit for every fault. We implement this virtual network through reconfiguration, limiting the amount of extra resources needed. This is a routing problem, as during the ATPG cycle the only aspects of the FPGA that have to be reconfigured are the routing resources and specifically, only the configuration cells for all the multiplexers in the routing switch boxes and the connection boxes.

In Figure 2 we show a traditional FPGA tool-flow (a) and the parameterised configurations flow (b), that uses runtime reconfiguration. In order to create a test set, traditionally the

initial design is injected with a single fault and synthesised, mapped, placed and routed on the target FPGA device. Then, a bitstream is generated that can be programmed into the FPGA. Normally, if a fault is observed by a different output value than the correct one, the test set is stored and then the device is reconfigured for another fault. Therefore, for each possible fault, the device needs to be reconfigured. The process is repeated until a test set is created for a large enough set of faults. At this point, we can observe that this process requires significantly long time. In order to reduce the time, a method can be used that creates multiple instantiations of the same design, each one implemented with different faults. This technique doesn't need the time consuming reconfigurations, however, it introduces area overhead and makes it practically impossible to be used for large designs.

In order to use the parameterised reconfiguration tool flow for our ATPG solution, we have designed a two-stage flow similar to the one in figure 1. Our tool, during the offline part uses fault injection with a new (instrumentation-based) technique and adds faults to each possible fault location, creating the virtual multiplexer network. So, our tool starts from a netlist and modifies it, by virtually adding extra hardware that represents a new fault. However, as every fault is annotated as a parameter, say for example fault *X*, the tool creates a generic bitstream (the PConf) that represents all circuit changes needed for every specific fault in a single bitstream, reducing the online changes needed to reflect a new fault to a simple Boolean function evaluation and a reconfiguration of the FPGA for the obtained new bitstream. This parameterized boolean bitstream can be rapidly evaluated for a specific fault automatically with the parameterised configurations tool flow. The design is never recompiled, only reconfigured.

A. The offline tool flow

In more detail, the new design (including all faults as parameters) has to pass all the typical compilation steps, namely synthesis, technology mapping, packing, placement and routing, similar to the tool flow shown in Figure 2(a). However, in our approach, the design has to pass each of the above steps only once. This computationally intensive stage is the offline stage of the tool flow. This section describes mostly the changes needed in each step of the offline part.

1) Synthesis: The design can originally be described in various HDLs, such as VHDL/Verilog. The synthesis step can be performed by any tool that is able to extract a BLIF format. At this point the design is ready for fault injection.

2) Parameterised Fault Injection: In this work, the technique focuses on ATPG and we assume that the injected fault set is either optimised, or it can be optimized by existing techniques such as fault dropping and fault collapsing. The fault model that is used (single stuck-at fault model) is widely applied within the testing community and an easy-to-implement method in order to introduce faulty behaviour
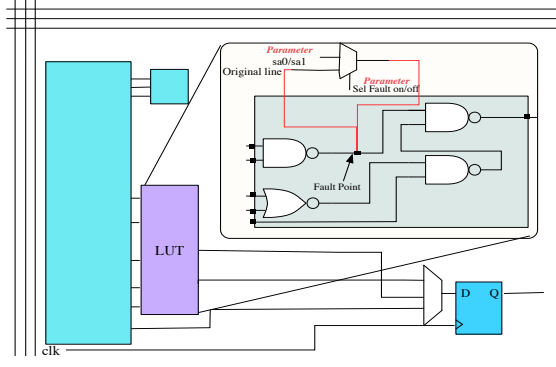
Figure 3.   Fault Emulation with parameterised fault injection

in the circuit. However, our methodology is not limited to this model alone.

The faults are added into the design at every possible fault location in such a way that after the new modifications, the new description remains synthesizable. The solution is to add multiplexers into each fault point to introduce a logic one or zero in order to mimic such fault, as shown in Figure 3. Our tool reads the netlist and locates all the possible fault checkpoints, i.e. locations where faults need to be inserted. The selection signals are annotated as *parameters*, as they will change (but less frequently than the other signals) depending on the type of fault and whether or not the fault should be injected. In our approach, we basically add logic (multiplexers) without adding more LUTs because they are depending only on parameters and can hence be implemented in the reconfiguration resources. So we basically have almost the same size as for the original circuit but now for an extended circuit with all faults injected.

3) TCON Technology Mapping: During technology mapping, the parameterized Boolean network generated by the synthesis step is not directly mapped onto the resource primitives available in the target FPGA architecture, but intermediately on abstract primitives that introduce and allow the reconfigurability of the logic and routing resources.

4) TPaR: Next, the Tunable Place and Route tool (TPAR) places and routes the netlist and performs packing, placement and routing with the algorithms TPack, TPlace and TRoute. These algorithms can enable routing of circuits where their routing resources can be reused during the fault emulation and drastically reduce the area usage. At the end of the computationally intensive offline stage the TPaR creates a *PConf*, a *virtual intermediate FPGA configuration* in which the bits are Boolean functions of the parameters.

B. The online tool flow: Test set generation cycle

Fast test set generation is essential. It is faster to generate random test inputs and select a viable test by emulation, than to effectively search for a test that detects the fault

in simulation. So, we use a Linear Feedback Shift Register (LFSR) to generate random input vectors. Both the initial circuit output and the fault-injected one have to be compared (with a XOR gate) for every different input set. If a fault is detected (by a difference in both output vectors) the input vector is stored as a test vector that detects the specific fault at hand. If the required fault coverage is not yet achieved, a new fault is chosen and the FPGA is reconfigured to match the new fault. The stored vectors form the test set. Details about its flow are shown in Figure 4. The exact details of how the reconfiguration is done during the online part of the tool flow is out of scope of this paper. We have not yet implemented this part, but the overhead of the reconfiguration for every fault is comparable but always smaller than the overhead we would have in the traditional reconfiguration based fault emulation, as we have less bits to reconfigure than in the original case. However, the big gains of our method lie in the offline part (which in other cases would also be online and very time-consuming or produce a circuit with a very large area).
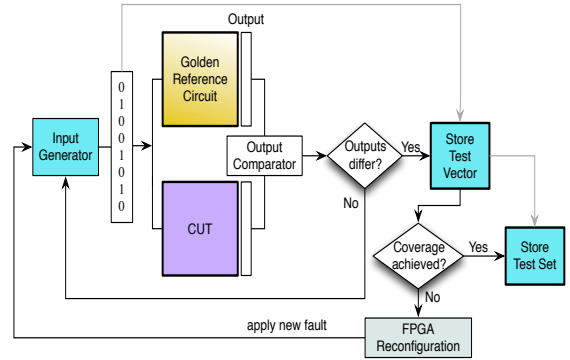

Figure 4.   The online tool Flow

C. Implementing the new ATPG tool flow

We adapted the VTR tool (a standard FPGA tool flow) [7] in such way, that it can be used as an automatic tool that injects faults into a design and produces a boolean bitstream (a bitstream that contains Boolean functions depending on the fault parameters). The VPR tool (part of VTR) has been extended so that it can support the parameterised configurations tool flow. The new version is able to reconfigure the FPGA's logic and routing resources. For this purpose, new modules have been created that can support the new design (with the injected faults).

Figure 2 shows the original VTR flow, as well as the adapted version. As mentioned in Section III-A, synthesis doesn't need to be modified. Then, a new step is added in the flow, fault injection. Afterwards, TCONMap is executed, which is integrated into a version of the original mapping tool [2]. The output file now is used as an input for VPR.

Golden	Conventional	Proposed(1)	Proposed(2)	Proposed(3)
24	237	50	41	25
39	346	133	106	45
124	861	302	253	126
137	869	337	289	144
129	1049	220	210	130
148	1027	400	334	153
149	1010	408	343	153

Table I

AREA RESULTS IN #LUTs: THE FIRST COLUMN IS FOR THE FAULT-FREE
DESIGN, THE OTHER COLUMNS CONTAIN ALL THE FAULTS. FOR THE
PROPOSED METHOD, WE DISTINGUISH MAPPING WITH PARAMETERISED
CONFIGURATION OF (1) LOGIC, (2) LOGIC & ROUTING, (3) LOGIC &
ROUTING WHERE SAF IS A PARAMETER AS WELL.

#LUTs	Golden	Proposed	Conventional
24	3	3	20
39	3	3	14
124	5	5	25
137	5	5	25
129	10	10	61
148	3	3	18
149	3	3	18

Table II

DEPTH COMPARISON BETWEEN THE INITIAL CIRCUIT AND THE FAULT
INJECTED VERSION MAPPED WITH 6-INPUT LUTs, WITH PROPOSED
MAPPER AND THE CONVENTIONAL ONE (ABC), RESPECTIVELY.

Figure 2 (b) shows all the stages from Verilog to a boolean bitstream representing faults depending on parameters.

IV. EXPERIMENTAL STUDY

This section presents the time overhead and the area impact of the proposed method. We have used synthesised MCNC benchmarks for our experiments. Despite the fact that we have used small benchmarks, we will also discuss the scaling behaviour and show that we can have interesting results also for larger designs. Since our tool follows the flow of VTR, it targets theoretical architectures. However, the results are representative for designs implemented in the LUT-based configurable logic of commercial FPGAs as well. In our experimental setup, our tool adds MUXs that can introduce stuck-at fault logic in all possible fault locations, as was described in Section III-A. Each MUX now describes one possible fault at one specific location. For each fault, the FPGA is reconfigured only once, but the FPGA is used multiple times to run all tests until one test is found that detects the fault. After that, the FPGA is reconfigured for a new fault.

A. Area usage

For our experiments, we compared the area of the fault injected circuit to the reference (fault-free) design. Four possible structures are evaluated. The first proposed structure is a normal MUX implementation (without any dynamic reconfiguration), mapped with a conventional mapper. The second structure is an implementation, where only the logic (functional) resources are dynamically reconfigured. The third approach is a structure where we reconfigure both logic and routing resources and finally, a fourth approach, where the stuck-at fault signal was also identified as an infrequently varying input, alongside dynamic reconfiguration of logic and routing resources. All these approaches are compared in Table I, with the initial, fault-free design (golden).

In the first approach, the lack of dynamic reconfiguration causes a massive area overhead (8 times more area), making this approach infeasible for large designs (as the extra area needed is proportional to the number of faults). A step

forward is mapping of the design with dynamic reconfiguration of its logic resources only (second approach). Here, we can see an area reduction compared to VTR's conventional mapper (up to a factor of 4.7). However, there is still a significant area overhead, compared to the golden circuit. The next approach applies dynamic reconfiguration of the FPGA's logic and routing resources, reducing the area even more. Finally, the last approach, builds upon approach 3. Now, the stuck-at fault signal is also annotated as a parameter. This results in the minimal area overhead, (needs up to 8 times less LUTs than the conventional method, and is only 3% larger than the initial design). Moreover, the problem can scale very well, making the technique feasible for larger designs.

B. Critical Path Delay

Our mapper reduces the critical path delay of the added functionality for the faults by reducing the number of lookup tables and the routing infrastructure on the critical path. From Table II, we can observe that the logic depth (inversely related to clock speed) of the design remains constant after the fault injection and the use of our mapper. This is very different from the conventional fault injection technique that introduces MUXs also in the critical path. In fact, the logic depth decreases with a factor of 5 to 8 in our ATPG method, compared to the conventional method.

C. Timing Impact Estimation

The conventional approach as described above has a prohibitive area overhead for large designs. However, as no reconfiguration is needed, it does not have any reconfiguration time overhead. In this section we discuss the reconfiguration time overhead of our method.

The runtime overhead depends on the number of times the emulator needs to be reconfigured and on the reconfiguration overhead, the time to evaluate the PConf and to reconfigure the bits that changed. The frequency of reconfiguration depends on the ATPG method. It needs to be reconfigured when a new fault needs to be activated. Therefore, the time overhead can be expressed as the single specialization time (for specializing the FPGA once) multiplied by the number of times a new fault will be activated. For large designs, a

lot of random tests are needed on average before a fault is detected. As long as the time needed for the evaluation of all these tests is significantly larger than the single specialisation time, our technique will have a relatively low time overhead. The single specialization time depends on the evaluation time and the time required for reconfiguration.

The evaluation time is needed to evaluate the Boolean functions in the parameterized configuration produced by the offline generic stage of the TCON tool flow. The time needed for one parameterised reconfiguration is highly dependent on the complexity of the boolean function, and needs maximum 50 μs. Thus, each parameterised configuration can be 3 orders of magnitude faster than a full reconfiguration, which is typically 176 milliseconds for a Xilinx Virtex-5 FPGA. Also, assuming the FPGA design runs at 400 MHz (which is quite fast for an FPGA implementation) and the test vector generation loop in figure 4 can be executed in 4 clock ticks (which requires a fully pipelined design), the 50 μs overhead corresponds with the time needed to perform 5000 tests on the FPGA fabric. This is a reasonable number for large designs. So for larger designs, the overhead becomes smaller relative to the test set generation time.

Besides the methods that have area overhead but no reconfiguration overhead, there is also a common technique that does not have an area overhead but requires execution of the entire tool flow online [6]. In this case, bitstream manipulations are performed on the fly, which takes a lot of time. Since they have to reconfigure the entire chip for every different fault, they would need 176 ms + 169.738 ms for each bitstream manipulation. Comparing this number with our online tool flow that takes 176 ms + 50 μs we are about twice as fast. However, the online time needed is only 50 μs in our case and thus orders of magnitude faster. This shows that the proposed method can introduce minimal time overhead as well. Also, our tool flow automatically produces the bitstream from simple additions to the HDL code, without the need for low-level bitstream manipulations to be done by the designer (as in the other approach).

V. Conclusions and future work

Fault emulation provides numerous advantages over fault simulation techniques, but, up to now, at a considerable cost. This paper proposes a new technique, where a test set can be generated efficiently (with minimal area overhead and minimal time overhead at the same time) and thus enhance the time consuming testing procedure during the design flow. Moreover, first experimental results demonstrate the feasibility of the approach, as it obtains an area and delay reduction of a factor of 8, and operates within 3 orders of magnitude faster, compared to conventional methods. Various extensions to this work are planned, such as applying a wider range of fault models, different FPGA architectures and larger designs.

References

[1] Elias Vansteenkiste, Brahim Al Farisi, Karel Bruneel, and Dirk Stroobandt. TPAR: Place and route tools for the dynamic reconfiguration of the FPGA's interconnect network. *Computer-Aided Design of Integrated Circuits and Systems, IEEE Transactions on*, 33(3):370–383, 2014.

[2] Karel Heyse, Karel Bruneel, and Dirk Stroobandt. Mapping logic to reconfigurable fpga routing. In *Field Programmable Logic and Applications (FPL), 2012 22nd International Conference on*, pages 315–321. IEEE, 2012.

[3] Lörinc Antoni, Régis Leveugle, and Béla Fehér. Using run-time reconfiguration for fault injection applications. *Instrumentation and Measurement, IEEE Transactions on*, 52(5):1468–1473, 2003.

[4] C. Lopez-Ongil, L. Entrena, M. Garcia-Valderas, M. Portela, M.A. Aguirre, J. Tombs, V. Baena, and F. Munoz. A unified environment for fault injection at any design level based on emulation. *Nuclear Science, IEEE Transactions on*, 54(4):946–950, Aug 2007.

[5] L. Sterpone and M. Violante. A new partial reconfiguration-based fault-injection system to evaluate seu effects in sram-based fpgas. *Nuclear Science, IEEE Transactions on*, 54(4):965–970, Aug 2007.

[6] Mojtaba Ebrahimi, Abbas Mohammadi, Alireza Ejlali, and Seyed Ghassem Miremadi. A fast, flexible, and easy-to-develop fpga-based fault injection technique. *Microelectronics Reliability*, (0):–, 2014.

[7] Jason Luu, Jeffrey Goeders, Michael Wainberg, Andrew Somerville, Thien Yu, Konstantin Nasartschuk, Jonathan Rose, and Vaughn Betz. Vtr 7.0: Next generation architecture and cad system for fpgas. *ACM Trans. Reconfigurable Technol. Syst.*, 7(2):6:1–6:30, July 2014.

[8] Kwang-Ting Cheng, Shi-Yu Huang, and Wei-Jin Dai. Fault emulation: A new methodology for fault grading. *Computer-Aided Design of Integrated Circuits and Systems, IEEE Transactions on*, 18(10):1487–1495, Oct 1999.

[9] P. Civera, L. Macchiarulo, M. Rebaudengo, M. Sonza Reorda, and M. Violante. An fpga-based approach for speeding-up fault injection campaigns on safety-critical circuits. *J. Electron. Test.*, 18(3):261–271, June 2002.

[10] L. Entrena, M. Garcia-Valderas, R. Fernandez-Cardenal, A Lindoso, M. Portela, and C. Lopez-Ongil. Soft error sensitivity evaluation of microprocessors by multilevel emulation-based fault injection. *Computers, IEEE Transactions on*, 61(3):313–322, March 2012.

[11] Abílio Parreira, JP Teixeira, and Marcelino Santos. A novel approach to fpga-based hardware fault modeling and simulation. In *Design and Diagnostics of Electronic Circuits and Syst. Workshop*, pages 17–24, 2003.