# Soft Error Protection Verification via Smart Behavioral Simulation

***Abstract*—This paper presents a novel approach to verify the Single Event Upset (SEU) resilience of a given netlist based on smart behavioral simulation. One of the most common ways of protecting a circuit against SEUs is Triple Modular Redundancy (TMR), which consists of triplication of flip-flops (FFs) and voting. In our approach, netlists hardened using TMR are converted into a special graph representation, and are then analyzed in order to detect any issues (that might incur during optimization, place and route, etc.) in the TMR implementation. Our analysis does not require a testbench and can perform full, exhaustive coverage within less than an hour for typical designs. This is achieved by analyzing the circuit and splitting it into smaller submodules without loss of generality, instead of simulating the whole netlist at once. The methodology has been verified on a full implementation of a LEON2-FT processor, checking exhaustively its soft-error protections.**

***Index Terms*—Triplication verification, SEU Analysis, Behavioral simulation, netlist conversion, Graph Representation, Triplet identification**

## I. Introduction

Hardening circuits against radiation effects is a complex task. At high altitude or in space, without the protection of the earth's magnetic field and atmosphere, integrated circuits are exposed to radiation and heavy ion impacts that can disrupt the circuits' behavior.

This paper focuses on the so called Single-Event-Upset (SEU) effects, or soft errors, usually caused by the transit of a single high-energy particle through the chip. This particle can upset storage elements and change their value from zero to one or vice versa by modifying the charge at a storage node [1]. Protection against SEUs can be done in several ways, and in particular this work considers the protection strategy based on triple modular redundancy (TMR), consisting of the triplication of every storage element combined with majority voting logic [2].

This protection can be either inserted during high level design [3] or at a later stage by automatic netlist modification. Typically, after a new ASIC is produced for the space market, it undergoes a strict test campaign, including costly and time consuming radiation tests using particle accelerators. When a problem arises during a radiation test campaign, it is already-too late; the first prototype ASICs have been manufactured and the whole fabrication process needs to be rerun after fixing the netlist. Detecting any problems before fabrication is the key, therefore several software [4]–[7] and hardware-based [8] tools for fault injection and verification were proposed in the past. However, such tools either do not provide full coverage or require extremely long simulation and/or execution times.

This paper presents a novel way to analyze the SEU sensitivity of a given netlist by taking a gate-level netlist as input and executing a smart behavioral simulation, and verifying the correctness of the implemented countermeasures against SEUs. To perform a fast analysis, we use a *divide et impera* approach, transforming the input netlist into a directed graph representation and analyzing smaller subgraphs, without loss of generality. Results show that verifying TMR on a 40k gates netlist is possible within around half an hour on a standard PC. Another advantage over other traditional simulation/verification methods is that our approach does not rely on a testbench, allowing a full coverage test.

This paper is organized as follows: previous works on the subject are introduced in Section II; Section III details the algorithm together with necessary definitions, its implementation and its complexity; experimental results are shown in Section IV, and Section V draws some concluding remarks.

## II. Previous work

In the past, several different approaches have been proposed for design verification against soft errors. These approaches can be divided in two kinds: fault injection simulation and formal verification.

Fault injection simulators run a given testbench on the design under test (DUT), flipping either randomly or specifically targeted bits. The outputs of the DUT are then compared with a golden model running the same testbench, and discrepancies are reported. Fault injection simulators come in two different flavors: on the one side there are software-based simulators like MEFISTO-L [5] or SST [6] (which is based on Modelsim), that allow full observability and control of the simulated netlist. These tools are marred by extremely slow low-level simulation, requiring hours or days of simulation, without any guarantee of full coverage. On the other hand some tools use special hardware to speed up the simulation cycle, such as FT-Unshades [8], which uses partial reconfiguration of an FPGA to quickly introduce single bit-flips (simulating SEUs) without requiring modifications of the DUT. Although this provides a consistent speedup compared to the software based approach, it is still infeasible to run full verification of a given design, which requires the injection of bit flips in all possible Flip-Flops (FFs) at any possible time during the simulation. It is also worth noting that the results of these approaches strongly depend on the testbench used.

Formal verification against soft-errors was introduced by [9]: the idea is to merge a formal model of the DUT with a soft error model, proving a given set of properties on the

merged model. This requires a formal model of the DUT and a complete and exhaustive set of formally defined properties to be proven. In other words, the main issue of this formal approach is that the coverage is as good as the definition of such properties.

This work tries to overcome these limitations and provide full SEU protection verification of a TMR-based DUT with reasonable simulation time. The idea presented in this paper can be classified as a fault-injection simulation, but follows a different approach with respect to previous work: instead of trying to simulate the whole circuit at once and doing a timing accurate simulation, we focus on the behavioral simulation of small submodules, extracted by automatic analysis of the DUT internal structure, with the specific goal of detecting any FF/voter pairs that are susceptible to SEUs.

## III. Proposed Approach

The starting point of our analysis is a radiation hardened circuit, protected by triplication and voting (TMR [2]). Our objective is finding any FFs that are not adequately protected.

Starting from a given design with $n$ FFs, a naive testing approach for SEU-susceptible FFs would require testing all $2^n$ possible configurations, for all of the $m$ time instants of a given testbench. This would lead to an impractically long simulation time, as typical as systems consist of several thousand FFs. Our approach uses the properties of the DUT in order to split the whole system into smaller submodules. Those small submodules can be analyzed independently, allowing a full test to be carried out in a reasonable timeframe.

The DUT is initially converted into a directed graph representation, based on a post-synthesis gate-level netlist (e.g. an EDIF file or a verilog netlist). This graph representation consists of nodes (modeling logic gates) and edges describing wires or interconnections between gates. Using this representation, each FF $i$ in the DUT is selected, and the subgraph of FFs connected to $i$'s input is calculated. Then, all valid start configurations (see Definition 6) for the FFs belonging to such a subgraph are calculated, simulating all possible bit flip injections. For every injection, the result is compared to fault-free simulation: if the results differ, the FF $i$ is marked as susceptible to SEUs.

We implemented a prototype of the algorithm relying on some assumptions: the whole circuit is driven by only one clock and there are no loops inside logic without a storage element being involved. Furthermore, it is assumed that there are no signal conflicts inside the netlist (i.e., two-valued logic) and that there are no timing violations. Finally, we assume that all FFs have one data input, one clock source, and all the triplets have separate reset and set lines. For the sake of simplicity, Single Event Effects (SEEs) on the reset and set lines are not considered, but they could be analyzed in a similar way.

### A. Mathematical model

To convert the netlist describing the circuit into a graph, we need to introduce a special directed graph structure. The nodes of this graph have indexed inputs and are associated to a logic function and a value, as outlined in the following. We assume without loss of generality that every gate has just one output. Gates that have $n \neq 1$ outputs are converted into $n$ nodes having the same inputs, each representing one output. Taking this into account the netlist can be easily converted into a directed graph structure

*Definition 1:* A *circuit graph* $G$ is defined as a tuple $\{V, E, S, F\}$, where:

- $V$ is a set of nodes (representing logic gates)
- $E \subseteq V \times V \times \mathbb{N}_0$ is a set of edges (representing interconnection wires)
- $S \subseteq V \times \{0, 1\}$ is a set of values (representing the node values)
- $F \subseteq V \times \mathcal{T}$ is the set of logic functions associated to each node, where $\mathcal{T}$ is the set of computable boolean functions

Every node $v \in V$ has 1 output and $num\_inputs(v) \subseteq \mathbb{N}_0$ inputs. The set of valid input indices for a node $v \in V$ is given by

$$N_v = \{1, ..., num\_inputs(v)\}$$

An edge $e = (x, y, i) \in E$ with $x, y \in V$ and $i \in N_y$ represents a connection from node $x$ to the input $i$ of node $y$. Assuming that the input circuit is free of driving conflicts, the circuit graph fullfills the property:

$$\forall v, w, x \in V, \forall i \in N_v : \\ v \neq w \wedge (w, x, i) \in E \implies (v, x, i) \notin E$$

which means that any given input of a node is connected to a single node output. We also assume that there are no unconnected inputs in the circuit, which translates to the property:

$$\forall x \in V, \forall i \in N_x, \exists w \in V : (w, x, i) \in E \tag{1}$$

To describe the algorithm, we need to define predicates that represent node properties.

*Definition 2:* The set of *direct predecessors* of node $x$, i.e. the set of nodes with a direct connection from their output to one of $x$ inputs is defined as:

$$pre(x) = \{w \mid \exists i \in N_x : (w, x, i) \in E\}$$

*Definition 3:* Let us define the predicate $is\_ff$ for a given node $x \in V$, which determines if $x$ represents a FF:

$$is\_ff(x) = \begin{cases} true & \text{if } x \in V \text{ is a FF or in-/output node} \\ false & \text{else} \end{cases}$$

For the sake of simplicity, top-level in-/outputs are considered as FFs with no inputs. The set of nodes that represent FFs is:

$$V_{FF} = \{x \mid \forall x \in V, is\_ff(x)\}$$

*Definition 4:* We define the set of nodes which are directly and indirectly connected to the inputs of a given node $x \in V$ as

the *smallest* set $pre\_ffs(x)$ for which the following properties hold $\forall w \in pre(x)$:

$$is\_ff(w) \implies w \in pre\_ffs(x)$$
$$\neg is\_ff(w) \wedge v \in pre\_ffs(w) \implies v \in pre\_ffs(x)$$

Having defined the FFs as just having one input (see Section III) we can define the driving node for a given FF as

*Definition 5:* A *driver* for FF $x \in V_{FF}$ is defined as:

$$driver(x) = \{y \mid (y, x, 1) \in E\}$$

Finally, we need the operators to compute the values associated to each node:

*Definition 6:* The value of a node $x \in V$ is given by the $eval$ operator, defined as:

$$eval(x) = \begin{cases} eval_{FF}(x) & \text{if } x \in V_{FF} \\ eval_L(x) & \text{else} \end{cases}$$

where $eval_F F$ returns the value stored in FF $x$:

$$eval_{FF}(x) = \{a \mid (x, a) \in S\}$$

and $eval_L$ computes the value of logic (i.e., non FF) nodes, which depends on the node input values:

$$eval_L(x) = \{f(eval(y_1), ..., eval(y_n)) \mid (x, f) \in F, y_i \in pre(x)\}$$

We also define the *configuration* of a set of FFs $x_i \in V_{FF}$ as

$$config(x_1, ..., x_n) = (eval(x_1), ..., eval(x_n))$$

A configuration $config(x_1, ..., x_n)$ is defined as *valid* when two FFs driven by the same logic value share the same value for all configurations:

$$\forall x_1, ..., x_n \in V_{FF}, \forall i \in N_{x_i}, \forall j \in N_{x_j} : driver(x_i) \equiv driver(x_j) \implies eval(x_i) = eval(x_j)$$

with $\equiv$ being defined as functionally identical (see Definition 7).

### B. Triplet identification

To determine a useful set of valid configurations for a subgraph, it is necessary to identify which FFs are triplicated, as all the FFs belonging to a triplet have to share the same value. However, if the gate naming scheme of a given netlist cannot be relied upon, this is not an easy task. A base assumption for triplet identification is that all triplicated FFs are driven by the same source. An algorithm based on this fact is able to find most triplets, but this simple mechanism is not always sufficient for more complex netlists.

During synthesis, netlists are often optimized in a way that voids this property. Figure 1 shows an example: Voter 2 was partially duplicated using other logic elements, with T2_OR and T3_NOT delivering the same values for all configurations of T2_FF_*, thus leaving T1_FF_2 with a different set of inputs with respect to the other members of the triplet. The synthesizer introduces this redundancy for delay optimization, place and route constraints, etc. Therefore, we assume that two

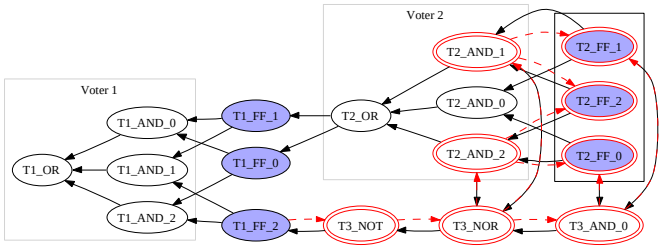


Fig. 1. Voter after optimization

FFs belong to one triplet(-group) if they are both driven by *functionally identical* nodes.

*Definition 7:* Two nodes $x_1$ and $x_2$ are functionally identical ($x_1 \equiv x_2$) if $pre\_ffs(x_1) = pre\_ffs(x_2)$ and $eval(x_1) = eval(x_2)$ for all possible configurations of $pre\_ffs(\cdot)$.

The test of all possible configurations for functionally identical inputs might be impractical, as it grows with $2^{pre\_ffs(x)}$. However, wrong triplet identification affects the verification of TMR protection only with the reporting of *false positives*, i.e. reporting a faulty triplicated structure when there is not one.

```
   function: functionally_identical(x,y)
   input   : nodes x, y ∈ V
   output  : number of matching configurations ∈ ℕ0
1  if driving_ffs(x) ≠ driving_ffs(y) then
2  |  return 0;
3  end
4  mark_graph(x);
5  (v1, ..., vk) ← find_marked(y, x);
6  count ← 0;
7  foreach c ∈ config(v1, ..., vk) do
8  |  for i ← 1 to k do
9  |  |  value(vi)← ci;
10 |  end
11 |  if eval(x) = eval(y) then
12 |  |  count ← count + 1;
13 |  end
14 end
15 return count;
```

**Algorithm 1**: `functionally_identical(x,y)`

Therefore, we propose a heuristic algorithm in three steps. The first step checks the sets of driving FFs for equality (see Algorithm 1, lines 1-3) before starting from $x$ (Figure 1,$T1\_FF\_2$) and traversing the graph depth first, marking all visited nodes (shown as a second circle in Figure 1), until a FF is visited and marked (Alg. 1, line 4).

In a second step the algorithm starts again from $y$ (Figure 1, T1_FF_1) and traverses the graph until reaching a marked node. If an unmarked FF is traversed, this shows that $x$ and $y$ are not functional identical[1] in the same clock cycle, and the algorithm aborts. After terminating successfully, the algorithm returns the set of marked nodes (Alg. 1, line 5). For the

[1]assuming no FFs were duplicated during optimization

example in Figure 1 this would be T2_AND_1, T2_AND_2, T2_FF_2.

The third step verifies that all configurations for this set have the same values for $x$ and $y$. This is done by assigning all possible configurations to this set (Alg. 1, lines 7-10) and evaluating the subgraph for $x$, $y$ to compare the results (line 11). Checking all possible configurations, as opposed to only valid ones, might result in functionally identical nodes not being identified. Instead of drawing a sharp yes/no conclusion, the number of matching configurations is compared for all possible triplet allocations, and the best one is used to assign the FFs. In other words, the nodes $x_i$ and $x_j$ belong to the same triplet(-group) for which $i$ and $j$ ($i \neq j$) result in the largest number of matching configurations.

It is worth noting that the worst case scenario for this fast heuristic, i.e. when all FFs are reported as false positives, is when both subgraphs share only the driving FFs and the whole subgraph is duplicated. This is unlikely to happen when analyzing real world netlists, because synthesizers optimize away most redundant parts and introduce redundancy only in rare cases. For the designs used in this work, the non-shared subgraph size is typically less than nine gates as shown by Figure 3.

### C. Simulation Algorithm

As stated in Section III the input of our algorithm is a radiation hardened circuit protected by triplication. Before starting the analysis, we optimize our description by removing (for our analysis) unnecessary elements as one-to-one buffer gates. This is done during netlist parsing. As such buffers do not manipulate the logic value of a signal; it is easy to see that the logic functions are not changed when they are removed.

If the TMR implementation were working correctly, a single bit-flip in one FF should not cause another FF to change its value. If a faulty triplicated FF/voter pair exists, there is at least one FF whose value can be changed by a single bit-flip in another FF. This is true only if the configuration before the bit-flip injection was a valid configuration. The algorithm tries to find such FFs, and if none is found, TMR is correctly implemented.

The main idea of the test algorithm is that complexity can be reduced by checking only small submodules instead of the whole system. In order to do this, we observe that a bit-flip in one FF can only distribute to the next FF during the current clock cycle. It is then possible to determine the set of all FFs which could potentially influence a given FF $x \in V_{FF}$, i.e. $pre\_ffs(x)$.

The algorithm takes each FF $x_i$ and determines the set of FF that are connected to it via logic only (no memory elements), and tests every possible bit flip for every possible valid configuration. If any of these bit flips is able to change $x_i$ stored value, then the algorithm detected a fault in the TMR implementation. More formally, Algorithm 2 describes this behavior in pseudocode (where `abort` interrupts execution and shows a message to the user). As the analysis has to be performed for all $x \in V_{FF}$, simulation times might be excessively long. To reduce runtime, this algorithm has to be extended to handle large sets of driving FFs $(y_1, ..., y_k)$. If the number of elements $t = |pre\_ffs(x)|$ in such a set exceeds a given threshold, the graph will be split into smaller subgraphs until the threshold is reached, as outlined in Section III-D.

```
   function: analyze(x)
   input   : a node x ∈ V
 1 (y_1, ..., y_k) ←pre_ffs(x);
 2 foreach valid c ∈ config(y_1, ..., y_k) do
 3     for i ← 1 to k do
 4         value(y_i)← c_i;
 5     end
 6     init_value ← eval_FF(x);
 7     foreach 1-bit mutation c' of c do
 8         for i ← 1 to k do
 9             value(y_i)← c'_i;
10         end
11         mut_value ← eval(x);
12         if mut_value ≠ init_value then
13             abort(FF x sensitive to SEUs);
14         end
15     end
16 end
```

**Algorithm 2**: `analyze(x)`

### D. Splitting algorithm

Analyzing typical designs with the proposed algorithm showed that the majority of FFs are driven by a very small set of FFs $pre\_ffs(x)$ (typically less than 9, see Figure 3). However there are a few FFs that are driven by a large number of FFs (for some designs 500 or more). Those subgraphs cannot be analyzed directly as they require $2^n$ configurations to be evaluated, and heuristics have to be devised.

A naive approach would use "divide et impera", splitting every node where $|pre\_ffs(y_i)| > threshold$, starting from the FF to be analyzed. This approach works for most FFs but fails if the synthesizer merged a voter with other logic during optimization. As an example, a 3-OR gate of a voter might be merged with a following OR gate into a 4-OR gate. Splitting could break the voter and result in a false positive alert.

Let $child_i$ be the nodes connected to the inputs of node $x$. To avoid breaking voting logic, instead of splitting using the threshold only, the originating node is kept and the subgraphs for the nodes $child_i$ with $|pre\_ffs(child_i)| > threshold$ are replaced by dummy input nodes (Alg. 3, lines 3-7). Every Node $child_i$ is tested recursively according to Algorithm 2 with the divide et impera approach.

Afterwards, all possible bit configurations are assigned to the dummy inputs connected to $x$ (lines 12-14). Analyzing $x$ for such configurations ensures that $x$ is tested for all possible substates previously generated by the removed nodes $child_i$.

It is worth noting that this heuristic relies on the fact that synthesizers tend to keep the voting logic close to the originating FFs, and therefore splitting subgraphs with a large number of inputs *usually* does not result in voters to be broken.

```
function : split_analyze(x)
input    : a node x ∈ V
split_required ← false;
foreach child ∈ pre(x) do
    if |pre_ffs(child)| > THRESHOLD then
        replace_input(x, child, dummy);
        split_analyze(child);
        split_required ← true;
    end
end
if split_required then
    (d_1, ..., d_k) ← get_dummynodes(x);
    foreach c ∈ config(d_1, ..., d_k) do
        for i ← 1 to k do
            value(d_i)← c_i;
        end
        analyze(x);
    end
else
    analyze(x);
end
```

**Algorithm 3**: `split_analyze(x)`

However, it cannot be excluded that some voting logic might be broken, resulting in some rare false positive alerts (see Section IV). This will *never hide any SEU sensitive parts*: if TMR is not properly implemented, it will be detected. In case the algorithm reports a SEU-sensitive FF, testing with a higher threshold value or manual inspection can identify if it represents a false positive.

### *E. Algorithm complexity analysis*

Given $m = |V|$ and $n = |V_{FF}|$, being the total number of gates and FFs, respectively, a naive exhaustive search would result in $2^n$ possible FF configurations to test, requiring $O(m2^n)$ node evaluations.

Determining a subgraph to be analyzed for every node $x \in V_{FF}$, gives $n$ subgraphs to verify. Using the properties presented in Section III-C, the algorithm has to check $p_x = |pre\_ffs(x)|$ FFs, with typical designs showing that in general $p_x \ll n$. As described in Section III-C, the algorithm limits $p_x$ to a given threshold $t$ by splitting the graph into subgraphs. Therefore there are less than $2^t$ valid configurations we have to evaluate for every subgraph (assuming FF triplication, we expect less than $2^{\frac{t}{3}}$ valid configurations). As we are testing one bit-flip at a time, we need to perform $t$ injections on every valid configuration. Obviously, the number of subgraphs obtained after splitting and their sizes cannot exceed the total number of gates $m$, resulting in less than $n \cdot 2^t \cdot t \cdot m$ subgraph evaluations. Overall, the algorithm performs $O(nm^2)$ node evaluations, showing polynomial behavior and outperforming other exponential verification methods.

## IV. Experimental results

The algorithm presented in Section III-C was implemented as a C++ program called XXX[2]. The graph is obtained in two steps: first a given Verilog netlist is converted into an intermediate file format, which is then read and analyzed by XXX. This separation makes the parser independent from the main program, allowing easy development of parsers for different input files.

The graph itself was implemented in a custom structure, using pointers whenever possible and STL [10] maps, vectors, and sort algorithms to maximize speed. In order to be ASIC library independent, the parser is able to read library cell definitions and design netlists, and to map all custom ASIC cells to standard gates (AND, OR, ...). If the ASIC library makes use of non-standard cells, the parser and XXX can be easily enhanced. The tool requires no user input during runtime, and shows status information like the overall progress, which gate is being processed etc.

TABLE I
Runtime comparison between FT-Unshades and XXX with thresholds 15 and 21

| Testcase | # gates[a] | # FFs | FT-U[b] | XXX-15 | XXX-21 |
|---|---|---|---|---|---|
| resetgen | 648 | 30 | 8h | <1m | <1m |
| pci mas | 14379 | 453 | 5d 5h | <1m | 2m |
| pci tar | 13768 | 546 | 6d 7h | <1m | 10m |
| mctrl | 35357 | 1251 | 14d 11h | 1m | 1m |
| fpu | 66967 | 1437 | 16d 15h | 10m | 10m |
| amod | 87193 | 3303 | 38d 5h | 1m | 3m |
| iu | 147894 | 4224 | 48d 21h | 8m | 406m |
| pci | 190987 | 7974 | 92d 7h | 4m | 264m |

[a] Gatecount after mapping library to standard logic cells
[b] not exhaustive

The implementation was tested on the submodule netlists of a radiation hardened LEON2-FT processor [11]. Table I shows the results of our tests with a threshold of 15 and 21, and compares the runtime with the expected runtime of FT-Unshades [8]. All tests were performed on a 2.66GHz Intel Core Duo workstation. The runtime for the FT-Unshades test was calculated based on ideal assumptions with a testbench with 200000 clock cycles and injecting in every possible FF, Assuming 5ms runtime for each test. It is worth noting that this short testbench duration cannot cover all possible internal substates therefore resulting in a non exhaustive test. A testbench that covers all internal substates is hard or even impossible to find and the simulation time would be so high to render the analysis impractical. As shown, compared with the FT-Unshades toolchain, which introduces a consistent speedup with respect to PC based simulators, our approach is several orders of magnitude faster.

Comparing XXX to an exhaustive approach, for example for the pci submodule, we have that this module is verified in less than $7974 \cdot 2^{\frac{15}{3}} \cdot 15 \cdot 190987^2 \approx 1.3 \cdot 10^{17}$ node evaluations (threshold $t = 15$). A naive approach would require $190987 \cdot 2^{7974} \approx 4.9 \cdot 10^{2405}$ evaluations, showing that XXX provides orders of magnitude of speedup.

As the actual runtime of XXX depends on the choice of the threshold presented in Section III-D, we tested several threshold values to determine the speed of the algorithm. In

[2] name omitted for blind review

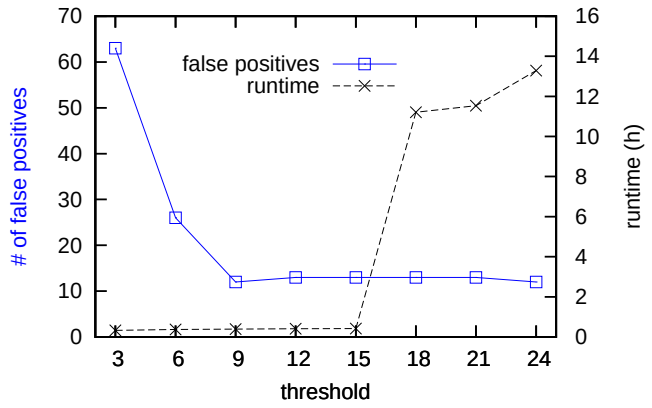


Fig. 2. Runtime and false positive count trend with increasing threshold

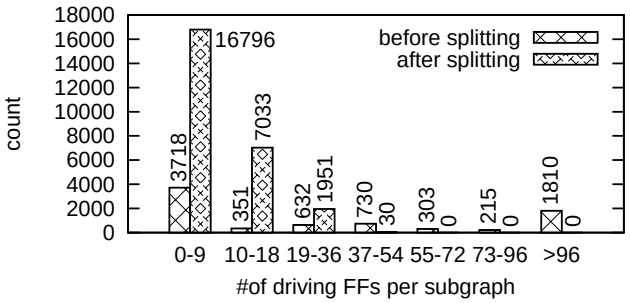


Fig. 3. Size classification of subgraphs before and after splitting

genera, smaller thresholds result in shorter runtimes with the drawback of more false positive alerts because of voters that have been broken during subgraph splitting. False positives have to be analyzed by manual graph inspection, or with other means. Figure 2 shows how overall runtime and number of false positives vary with increasing threshold, for all nine netlists of the LEON2-FT processor, with a total 19218 FFs and 570959 gates.

The sum of false positives for all nine given designs goes from 63 down to 12 The overall runtime goes from 19 minutes up to 13 hours. For a suggested threshold of 15, the runtime is around 25 minutes. Please note that the runtime strongly correlates to the internal structure of the design, especially the subgraph sizes, and therefore it is subject to large fluctuations among the designs.

To show the effectiveness of the subgraph splitting, the algorithm was tested on the nine netlists, logging the different subgraph sizes before and after splitting (with threshold=15). Figure 3 shows the results of this test. Before splitting, there are 1810 subgraphs that consist of $> 96$ driving gates. Assuming correct triplication this would result in more than $2^{32}$ valid configurations to be checked for each of those nodes, making the splitting heuristic an essential component of our approach. In fact, after splitting the situation is completely different: even though the splitting results in many more subgraphs to be checked, the subgraph sizes are much smaller. There are no subgraphs with more than 54 driving gates, giving no more than $2^{18}$ valid configurations

Finally, to show its fault detecting capabilities, XXX was verified on a netlist (module $iu$ in Table I) with broken voters. The netlist used for this test consists of 1408 triplets (4224 FFs). For the test run 898 triplets were automatically selected, and their voters manipulated by changing the voter function from $f(x_1, x_2, x_3) = (x_1 \wedge x_2) \vee (x_2 \wedge x_3) \vee (x_3 \wedge x_1)$ to $f(x_1, x_2, x_3) = x_1 \vee x_2 \vee x_3$. This should result in $n = 898 \cdot 3$ SEU-sensitive FFs being detected. XXX reported problems in $n = 899 \cdot 3$ FFs: all unprotected triplets plus one false positive.

## V. Conclusions & Future work

In this work we presented an algorithm to verify TMR implementation for given netlists. Performing exhaustive verification without the need of a testbench, this approach does not suffer under the quality and coverage of the given testbench as other solutions. First results show that verification of production-ready netlists can be carried out within few hours. To the best of the authors' knowledge, no other approach provides this kind of performance.

Future work includes replacing the actual simulation/injection step with the identification of triplets followed by formal verification of the correct propagation of flip-flop values through the voting logic. Another technique under study is to merge some of the ideas into future hardware-accelerated fault injectors. The triplet identification heuristic presented in this work, or the graph structural knowledge of independent subgraphs could be used to allow multiple injections at one time and thus reducing the test's runtime.

## References

[1] George C. Messenger and Milton S. Ash, *The effects of radiation on electronic systems*, 2nd ed. Van Nostrand Rinhold, 1986.

[2] C. Carmichael, *XAPP197: Triple module redundancy design techniques for Virtex FPGAs*, Xilinx Inc., July 2006. [Online]. Available: http://www.xilinx.com/support/documentation/application_notes/xapp216.p%df

[3] Sandi Habinc, "Functional Triple Modular Redundancy," Gaisler Research, Tech. Rep., 2002. [Online]. Available: http://www.gaisler.com/doc/fpga_003_01-0-2.pdf

[4] G. Kanawati and J. Abraham, "Ferrari: a flexible software-based fault and error injection system," *Computers, IEEE Transactions on*, vol. 44, pp. 248–260, 1995.

[5] J. Bou, P. Ptillon, and Y. Crouzet, "Mefisto-l: A vhdl-based fault injection tool for the experimental assessment of fault tolerance." IEEE Computer Society, 1998, p. 168.

[6] J. A. Maestro, *SST 2.0: User Manual*, Universidad Antonio de Nebrija, November 2006. [Online]. Available: http://www.nebrija.es/~jmaestro/esa/docs/SST-UserManual2-0.pdf

[7] K. K. Goswami, R. K. Iyer, and L. Young, "Depend: A simulation-based environment for system level dependability analysis," *IEEE Transactions on Computers*, vol. 46, no. 1, pp. 60–74, 1997.

[8] M. Aguirre, J.N. Tombs, V. Baena-Lecuyer, F. Muñoz, A. Torralba, A. Fernández-León, and F. Tortosa-López, "FT-UNSHADES: A new System for Seu Injection, analysis and diagnostics over post synthesis netlist," *MAPLD'2005, Nasa Military and Aerospace Programmable Logic Devices*, Sep. 2005.

[9] S. A. Seshia, W. Li, and S. Mitra, "Verification-guided soft error resilience," in *Proc. Design Automation and Test in Europe (DATE)*, April 2007.

[10] *Standard Template Library*, SGI. [Online]. Available: http://www.sgi.com/tech/stl

[11] J. Gaisler, "The LEON2 IEEE-1754 (SPARC V8) Processor," Gaisler Research, 2003. [Online]. Available: http://www.gaisler.com