Triple Modular Redundancy verification via heuristic netlist analysis

Giovanni Beltrame

Polytechnique Montréal, Montréal, Québec, Canada

ABSTRACT

Triple Modular Redundancy (TMR) is a common technique to protect memory elements for digital processing systems subject to radiation effects (such as in space, high-altitude, or near nuclear sources). This paper presents an approach to verify the correct implementation of TMR for the memory elements of a given netlist (i.e., a digital circuit specification) using heuristic analysis. The purpose is detecting any issues that might incur during the use of automatic tools for TMR insertion, optimization, place and route, etc. Our analysis does not require a testbench and can perform full, exhaustive coverage within less than an hour even for large designs. This is achieved by applying a divide et impera approach, splitting the circuit into smaller submodules without loss of generality, instead of applying formal verification to the whole netlist at once. The methodology has been applied to a production netlist of the LEON2-FT processor that had reported errors during radiation testing, successfully showing a number of unprotected memory elements, namely 351 flip-flops.

Subjects Computer Aided Design, Computer Architecture, Embedded Computing
Keywords Single event effects, Triple Modular Redundancy, Verification

Submitted 29 May 2015
Accepted 10 August 2015
Published 26 August 2015

Corresponding author
Giovanni Beltrame,
giovanni.beltrame@polymtl.ca

Academic editor
Mary Sheeran

Additional Information and Declarations can be found on page 16

DOI **10.7717/peerj-cs.21**

INTRODUCTION

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 Single-Event-Upsets (SEUs), or soft errors, usually caused by the transit of a single high-energy particle through the circuit. In particular, we consider single bit flips in memory elements embedded in logic, implemented as flip-flops. Protection against SEUs can be obtained in several ways, and in particular this work considers the protection strategy based on the triplication of the storage elements of a circuit, combined with majority voting (*Carmichael, 2006*), usually referred to as Triple Modular Redundancy (TMR).

TMR can be either implemented during high level design (*Habinc, 2002*) or at a later stage by automatic netlist modification. Typically, after a new radiation-tolerant ASIC is produced, it undergoes a strict test campaign, including costly and time consuming radiation tests using particle accelerators. When a problem linked to the radiation effects protection logic 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. Detecting this kind of problems before fabrication is key, therefore several software (*Kanawati & Abraham, 1995*; *Boué, Pétillon & Crouzet, 1998*; *Maestro, 2006*; *Goswami, Iyer & Young, 1997*) and hardware-based (*Aguirre et al., 2005*) tools for fault injection and protection verification have been proposed in the recent past. However, such tools usually are not designed to provide full SEU protection verification, and require extremely long simulation and/or execution times when attempting comprehensive fault injection and analysis campaigns. To the best of the author's knowledge, no commercial or academic tool providing TMR implementation verification is currently available.

This paper presents a novel way to verify the TMR implementation of a given circuit by executing a heuristic netlist analysis. Our goal is to verify that TMR constructs are insensitive to single bit flips (i.e., the logic is triplicated), and transients on clock or reset (i.e., there are no common reset/clock lines between redundant memory elements). To reduce execution time, we use a *divide et impera* approach, splitting the netlist in smaller submodules, 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. As this work is based on formal analysis, our approach does not rely on a testbench, allowing a full coverage test based solely on the device netlist.

This paper is organized as follows: previous works on the subject are introduced in 'Previous Work'; 'Proposed Approach' details the algorithm together with necessary definitions, its implementation and its complexity; experimental results are shown in 'Experimental Results', and 'Conclusions and Future Work' draws some concluding remarks.

PREVIOUS WORK

In the past, 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 (*Boué, Pétillon & Crouzet, 1998*) or SST (*Maestro, 2006*) (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, making them unsuitable for full coverage tests. On the other hand, some tools use special hardware to speed up the simulation cycle, such as FT-Unshades (*Aguirre et al., 2005*), which uses partial reconfiguration of an FPGA to quickly introduce 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 unfeasible to run exhaustive verification of a typical ASIC design in full, which would require 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 and how they can be interpreted strongly depend on the testbench used.

Formal verification against soft-errors was introduced by (*Seshia, Li & Mitra, 2007*): 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 verification of a TMR-based DUT with reasonable analysis time. The idea presented in this paper can be classified as a fault-injection simulation, but follows a different approach as compared to previous work: instead of trying to simulate the whole circuit at once and doing a timing accurate simulation, we focus on a behavioral, timeless, simulation of small submodules, extracted by automatic analysis of the DUT internal structure, with the specific goal of detecting any triplicated FF that is susceptible to the propagation of SEUs in the DUT.

PROPOSED APPROACH

The starting point of our analysis is a radiation hardened circuit, protected by triplication of storage elements and voting (TMR in *Carmichael (2006)*). Our objective is to verify that indeed all FFs are adequately protected, and no issues were introduced, for example, by synthesis or routing tools.

Starting from a given design with n FFs, a naive testing approach for SEU-sensitive FFs would require injecting faults in all 2^n possible FF configurations, for all of the m time instants of a given testbench. This would lead to an impractically long simulation time, as typical systems consist of several thousand FFs. Our approach performs a behavioral fault injection, splitting the whole system into smaller submodules, that can be analyzed independently, allowing full verification to be carried out in a reasonable timeframe.

These submodules are the *logic cones* driving each FF. A logic cone is a set of combinational logic bounded by FFs and I/O (see Fig. 1). To verify that no FF are sensitive to SEUs (therefore assuring the correct TMR implementation), it is possible to extract its driving logic cone, and perform an exhaustive fault injection campaign. This means that the FFs bounding the logic cone are injected single bit flips in all possible input configurations, comparing the output of the logic cone with its expected (i.e., fault-free) one. It is also necessary to verify that all the triplicated FFs have separate asynchronous reset and set lines, otherwise a transient on one of these lines might still cause a failure in the circuit. Testing all possible configurations for a logic cone means 2^{n_f} injections, with n_f the number of driving flip-flops making the analysis difficult or impossible for high n_f. When TMR is applied to each memory element, n_f increases by a factor 3 in each logic cone, and the cone itself is modified to account for the voting logic, as shown in Fig. 2. However, when a TMR implementation is present, each logic cone is driven by *triplets* of flip-flops and Fig. 2 shows how his restricts the number of input configurations that are actually valid for the cone, as all FF belonging to the same triplet share have the same value during normal operation of the circuit. The proposed algorithm, considering only *valid configurations* when performing fault injection, reduces the number of injections

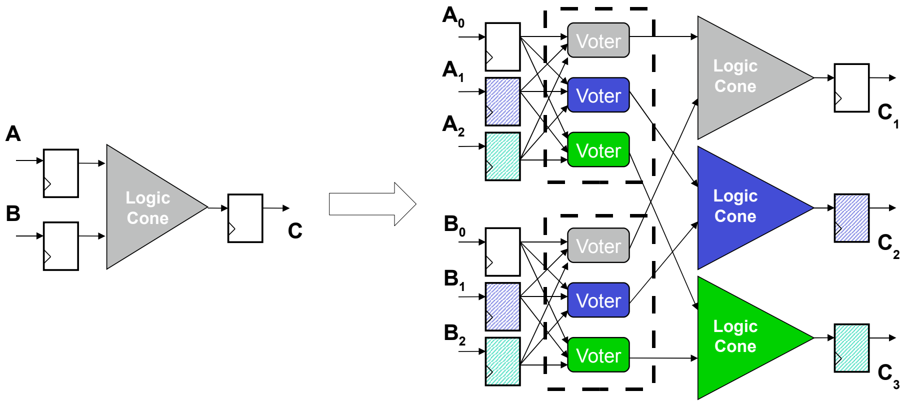
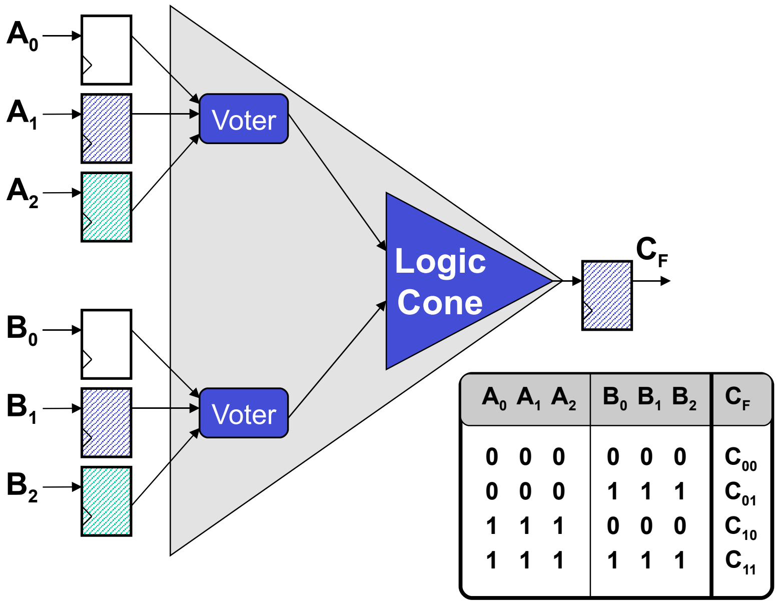


Figure 1 **A logic cone is a set of logic bounded by FFs and I/O.** When TMR is applied, each logic cone contains part of the voting logic.



A_0 A_1 A_2	B_0 B_1 B_2	C_F
0 0 0	0 0 0	C_{00}
0 0 0	1 1 1	C_{01}
1 1 1	0 0 0	C_{10}
1 1 1	1 1 1	C_{11}

Figure 2 A logic cone where FF triplets have been identified: the valid configurations are shown.

to $2^{n_f}(1+n_f)$, with n_f the number of driving FFs for each logic cone. This results in a considerable analysis speed-up: Fig. 3 shows the trend for the number of injections needed as n_f increases.

The methodology here presented relies on some assumptions: the whole circuit is driven by only one clock and there are no combinatorial loops. 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 and one clock source.

Mathematical model

To describe the algorithm, 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,

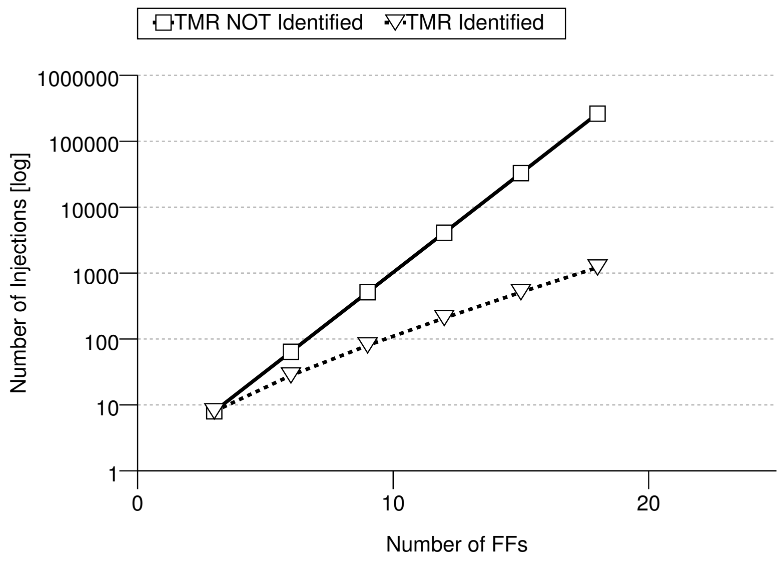


Figure 3 **Trend for the number of checks needed per for a logic cone with as the number of driving FFs increases.**

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

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

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:

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

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:

$$\forall x \in V, \forall i \in N_x, \exists w \in V : (w,x,i) \in E. \quad (1)$$

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:

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

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

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

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 assumed that each FFs has one input, 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:

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

where $eval_{FF}$ 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:

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

We also define the *configuration* of a set of FFs $\{x_1, x_2, ..x_n\} \in V_{FF}$ as

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

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

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

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

The proposed methodology is composed of 3 steps:

1. Triplet identification: determine all the FF triplets present in each logic cone
2. TMR structure analysis: perform an exhaustive fault injection campaign on all valid configurations
3. Clock and reset tree verification: assure that no FF triplet has common clock or set/reset lines

These steps are detailed in the following.

Triplet identification

To determine a useful set of valid configurations for a logic cone (here represented by 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, the gate naming scheme is usually insufficient. 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 4 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 FFs belong to one triplet 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 $pre\_ffs(\cdot)$.

Testing for functionally identical inputs requires *equivalence checking* (Thornton, Drechsler & Gunther, 2000) of the logic functions expressed by the nodes. For the sake of

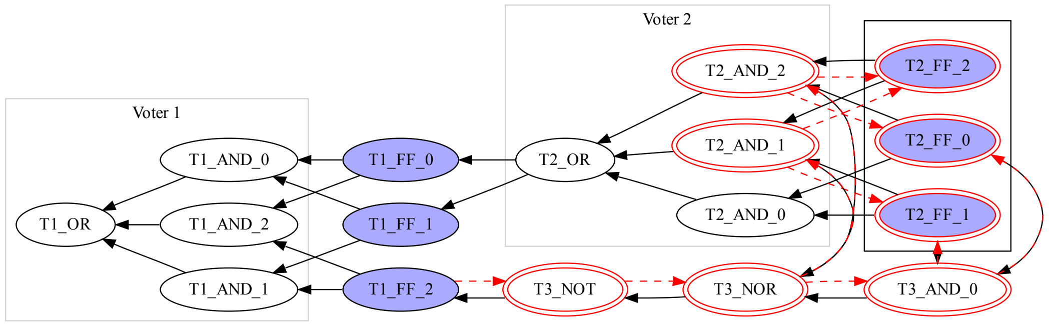


Figure 4 A sample graph with FF triplets and voters after optimization.

simplicity we implemented a simple checker that exhaustively compares all possible input configurations for the two nodes. Checking the equivalence between two nodes might be impractical, as the problem grows exponentially (roughly $2^{pre\_ffs(x)}$). 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 none exists. Therefore, we propose a heuristic algorithm in three steps.[1] Let us consider the example of Fig. 4, where Algorithm 1 is applied to $T1\_FF\_2$ and $T1\_FF\_1$ (x and y in the algorithm, respectively). Two sub-algorithms are needed:

[1] It is worth noting that other algorithms for functional equivalence checking can be used here.

Definition 8 $mark\_graph(x)$ traverses the graph starting at node x and marks all visited nodes, stopping at FFs. Its behavior is formalized in Algorithm 8.

Definition 9 $find\_marked(y,x)$ traverses the graph depth-first starting from y until a FF is encountered and returns all the nodes that were found as marked by x. Its behaviour is formalized by Algorithm 10.

The first step checks the sets of driving FFs for equality (lines 1–3) before starting from x (Fig. 4, $T1\_FF\_2$) and traversing the graph depth first, marking all visited nodes (shown as a second circle in Fig. 4), until a FF is visited and marked (line 4).

In a second step the algorithm starts again from y (Fig. 4, $T1\_FF\_1$) and traverses the graph until reaching a marked node. If an unmarked FF is traversed, this shows that x and y are not functionally identical[2] 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 example in Fig. 4 this would be $T2\_AND\_1$, $T2\_AND\_2$, $T2\_FF\_2$.

[2] Assuming no FFs were duplicated during optimization.

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 recognized. 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 for which i and j ($i \neq j$) result in the largest number of matching configurations.

```
function : functionally_identical(x,y)
input    : nodes $x, y \in V$
output   : number of matching configurations $\in \mathbb{N}_0$
if $pre\_ffs(x) \neq pre\_ffs(y)$ then
    return 0;
end
mark_graph($x$);
$(v_1, \ldots, v_k) \leftarrow$ find_marked($y, x$);
$count \leftarrow 0$;
foreach $c \in$ config($v_1, \ldots, v_k$) do
    for $i \leftarrow 1$ to $k$ do
        value($v_i$) $\leftarrow c_i$;
    end
    if $eval(x) = eval(y)$ then
        $count \leftarrow count + 1$;
    end
end
return $count$;
```

Algorithm 1: functionally_identical(x,y)

```
input : a node $x \in V$
input : a node $marker \in V$
last_visit($x$) $\leftarrow marker$ ;
if $is\_ff(x)$ then
    return;
else
    foreach $child \in$ pre($x$) do
        mark_graph($child, marker$);
    end
end
```

Algorithm 2: mark_graph()

```
input  : a node $x \in V$
input  : a node $marker \in V$
output : a set of nodes $(r_1, \ldots, r_n), r_i \in V$
if last_visit($x$) $\neq marker$ then
    return $\emptyset$;
else
    result_set $\leftarrow (x)$;
    foreach $child \in$ pre($x$) do
        rek $\leftarrow$ find_marked($child, marker$);
        result_set $\leftarrow$ result_set $\cup$ rek;
    end
    return result_set;
end
```

Algorithm 3: find_marked()

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 Fig. 7.

TMR structure analysis

Before starting the analysis, we optimize our description by removing non-relevant elements, as one-to-one buffer gates. 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 are 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., $pre\_f\,fs(x)$, i.e., the FFs driving x's logic cone.

The algorithm takes each FF x_i and determines the set of FFs that driving its logic cone, 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 4 describes this behavior in pseudocode (where `abort` interrupts execution and shows a message to the user). As the analysis

```
function : analyze(x)
input    : a node $x \in V$
$(y_1, \ldots, y_k)$ ← pre_f fs($x$);
foreach valid $c \in$ config($y_1, \ldots, y_k$) do
    for $i \leftarrow 1$ to $k$ do
        value($y_i$) ← $c_i$;
    end
    init_value ← $eval_{FF}(x)$;
    foreach 1-bit mutation $c'$ of $c$ do
        for $i \leftarrow 1$ to $k$ do
            value($y_i$) ← $c'_i$;
        end
        mut_value ← $eval(x)$;
        if mut_value $\neq$ init_value then
            abort(FF x sensitive to SEUs);
        end
    end
end
```

Algorithm 4: `analyze(x)`

has to be performed for all $x \in V_{FF}$, analysis times might be excessively long. To reduce runtime, this algorithm has to be extended to handle large sets of driving FFs $(y_1, \ldots, y_k)$. If the number of elements $t = |pre\_f\,fs(x)|$ in such a set exceeds a given threshold, the graph will be split into smaller subgraphs until the threshold is reached, as outlined in 'Splitting algorithm'.

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_$f\,fs(x)$ (typically less than 9, see Fig. 7). 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. It is worth noting that the count from Algorithm 15 is not what is used here as a threshold. Here we consider only the number of driving flip-flops, and not the number of matching configurations between two logic cones. 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.

```
   function : split_analyze(x)
   input    : a node $x \in V$
1  split_required ← $false$;
2  foreach $child \in$ pre($x$) do
3      if $|pre\_ffs(child)| > THRESHOLD$ then
4          replace_input($x$, $child$, $dummy$);
5          split_analyze($child$);
6          split_required ← $true$;
7      end
8  end
9  if split_required then
10     $(d_1, \ldots, d_k)$ ← get_dummynodes($x$);
11     foreach $c \in$ config($d_1, \ldots, d_k$) do
12         for $i \leftarrow 1$ to $k$ do
13             value($d_i$) ← $c_i$;
14         end
15         analyze($x$);
16     end
17 else
18     analyze($x$);
19 end
```

Algorithm 5: `split_analyze(x)`

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. 5, lines 3–7). Every Node $child_i$ is tested recursively according to Algorithm 4 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.

However, it cannot be excluded that some voting logic might be broken, resulting in some rare false positive alerts (see 'Experimental Results'). This will *never hide any SEU sensitive parts*: if TMR is not properly implemented, this will be detected. In case the

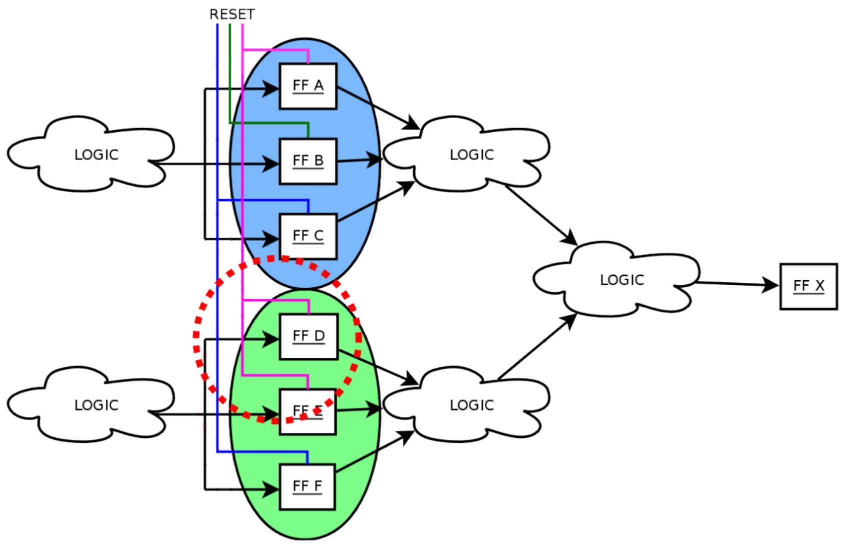


Figure 5 A shared reset line in a TMR triplet might cause issues in case of transients as two FFs might be affected simultaneously.

algorithm reports a SEU-sensitive FF, testing with a higher threshold value or manual inspection can identify if it represents a false positive.

Clock and reset tree verification

Verifying that the voters are correctly performing their task is not sufficient to guarantee that TMR structures are working. One also needs to show that transient errors on clock and reset lines are not affecting more than one FF at a time. Figure 5 shows how a shared reset line might result in SEU-like behavior, as a transient on the line affects more than one FF. This could happen if the FFs shared a common asynchronous reset or clock line: in this case, a bit flip might zero the entire triplet, or force the FFs to sample the wrong value. To guarantee that TMR structures are functioning, it is then necessary to rule out this possibility.

Using the detected triplets, it is possible to verify that FFs belonging to the same triplet do not share the same clock and reset lines. This is a simple structural analysis that does not require an heuristic to be performed.

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 'TMR structure analysis', the algorithm has to check $p_x = |pre\_f\ fs(x)|$ FFs, with typical designs showing that in general $p_x \ll n$. As described in 'TMR structure analysis', 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

Table 1 **Runtime comparison between FT-Unshades and InFault with thresholds 15 and 21.** Times in minutes (m), hours (h), and days (d).

Testcase	# gates[a]	# FFs	FT-U[b] time	InFault time (th-15)	InFault time (th-21)	FP[c]
Resetgen	648	30	8h	<1m	<1m	0
pci mas	14,379	453	5d 5h	<1m	2m	0
pci tar	13,768	546	6d 7h	<1m	10m	0
mctrl	35,357	1,251	14d 11h	1m	1m	0
fpu	66,967	1,437	16d 15h	10m	10m	6
amod	87,193	3,303	38d 5h	1m	3m	2
iu	147,894	4,224	48d 21h	8m	406m	3
pci	190,987	7,974	92d 7h	4m	264m	2

Notes.
[a] Gatecount after mapping library to standard logic cells.
[b] not exhaustive.
[c] False positives, same results for both thresholds.

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.

EXPERIMENTAL RESULTS

The algorithm presented in 'TMR structure analysis' was implemented as a C++ program called InFault. 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 InFault. 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 (*Silicon Graphics, 2000*) 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 InFault 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.

The implementation was tested on the submodule netlists of a radiation-hardened LEON2-FT processor (*Gaisler, 2003*). Table 1 shows the results of our tests with a threshold of 15 and 21, and compares the runtime with the expected runtime of FT-Unshades (*Aguirre et al., 2005*). All tests were performed on a 2.66 GHz Intel Core Duo workstation. Although FT-Unshades is not designed for full test coverage, Table 1 gives an idea of the performance of the algorithm over a brute force approach. It is worth noting that InFault does not need a testbench for providing results, since it performs a static analysis of the LEON2-FT gate-level netlist. However, to compare our results with FT-Unshades, we had to select a set of benchmarks. The runtime for the FT-Unshades test

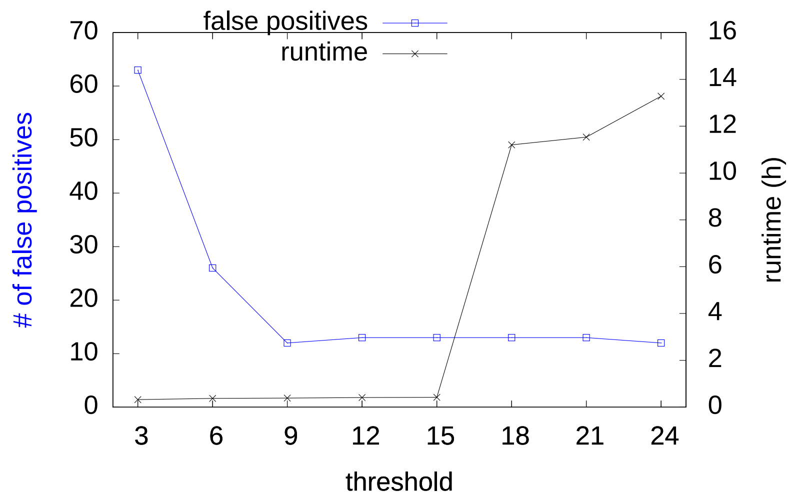


Figure 6 **Runtime and false positive count with increasing threshold.**

was calculated based on measurements on smaller tests, using a testbench with 200,000 clock cycles and injecting in every possible FF, assuming a faster-than-real 5 ms runtime for each test. These testbenches come directly from Gaisler Research, and are made with high code-coverage in mind. Therefore, they were considered as a good choice to stimulate every part of the processor. 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.

Comparing InFault 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 InFault provides orders of magnitude of speedup.

As the actual runtime of InFault depends on the choice of the threshold presented in 'Splitting algorithm,' we tested several threshold values to determine the speed of the algorithm. In general, 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 6 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 min up to 13 h. For a suggested threshold of 15, the runtime is around 25 min. Please note that the runtime strongly correlates to the internal structure of the design, especially the subgraph sizes, and therefore it is subject to 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

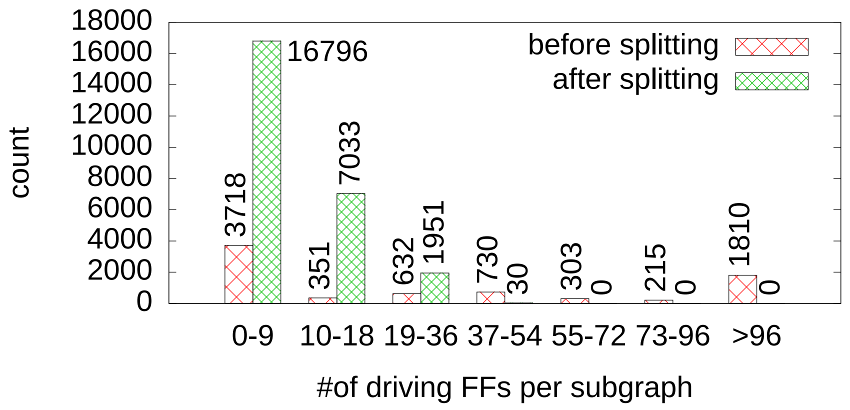


Figure 7 Size classification of subgraphs before and after splitting.

threshold = 15). Figure 7 shows the results of this test. Before splitting, there are 1,810 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.

It is worth noting that the results depend on the complexity of the circuit, since the splitting algorithm effectiveness varies by the number of driving FFs. InFault might not have the same performance with other processors with very complex multi-layer logic.

To show its fault detecting capabilities, InFault was verified on a netlist (module *iu* in Table 1) with broken voters. The netlist used for this test consists of 1,408 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. InFault reported problems in $n = 899 \cdot 3$ FFs: all unprotected triplets plus one false positive. Finally, the methodology was applied to a production netlist of the LEON2-FT processor that reported failures due to SEUs after manufacturing, during a radiation test campaign. The algorithm reported 351 common reset lines for triplicated FFs. An SEU on these lines is consistent with the behaviour shown by the processor during radiation testing. These errors were introduced by the automated tools in charge of inserting TMR structures in the LEON2-FT processor, during the optimization phase. After correcting the netlist, with the proposed methodology reporting no remaining issue, the processor was irradiated further. Radiation tests showed no additional problems due to the TMR implementation, so that we can reasonably assume that the 351 flip-flops were the cause of the initial testing issues.

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 from the quality and coverage of the given testbench as in other solutions. First results show that exhaustive TMR 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.

ACKNOWLEDGEMENTS

The author would like to thank Simon Schulz and David Merodio-Codinachs for the help provided in the development of InFault.

ADDITIONAL INFORMATION AND DECLARATIONS

Funding

The author received no funding for this work.

Competing Interests

The author declares there are no competing interests.

Author Contributions

- Giovanni Beltrame conceived and designed the experiments, performed the experiments, analyzed the data, contributed reagents/materials/analysis tools, wrote the paper, prepared figures and/or tables, performed the computation work, reviewed drafts of the paper.

Data Availability

The following information was supplied regarding the availability of data:

http://github.com/mistlab/infault.

REFERENCES

Aguirre M, Tombs JN, Baena-Lecuyer V, Muñoz F, Torralba A, Fernández-León A, Tortosa-López F. 2005. FT-UNSHADES: a new system for seu injection, analysis and diagnostics over post synthesis netlist. In: *MAPLD'2005, Nasa Military and Aerospace Programmable Logic Devices. Available at* *http://klabs.org/mapld05/abstracts/1014_aguirre_a.html*.

Boué J, Pétillon P, Crouzet Y. 1998. MEFISTO-L: a VHDL-based fault injection tool for the experimental assessment of fault tolerance. In: *Fault-Tolerant Computing, 1998. Digest of Papers.*

Twenty-Eighth Annual International Symposium on, vol., no., 23–25 June 1998. Piscataway: IEEE Computer Society, 168 DOI 10.1109/FTCS.1998.689467.

Carmichael C. 2006. *XAPP197: Triple Module Redundancy design techniques for Virtex FPGAs*. San Jose: Xilinx Inc. *Available at http://www.xilinx.com/support/documentation/application notes/xapp216.pdf*.

Gaisler J. 2003. The LEON2 IEEE-1754 (SPARC V8) processor. *Available at http://www.gaisler.com*.

Goswami KK, Iyer RK, Young L. 1997. Depend: a simulation-based environment for system level dependability analysis. *IEEE Transactions on Computers* **46(1)**:60–74 DOI 10.1109/12.559803.

Habinc S. 2002. Functional Triple Modular Redundancy. Technical report. Göteborg: Gaisler Research. *Available at http://www.gaisler.com/doc/fpga_003_01-0-2.pdf*.

Kanawati G, Abraham J. 1995. Ferrari: a flexible software-based fault and error injection system. *IEEE Transactions on Computers* **44**:248–260 DOI 10.1109/12.364536.

Maestro JA. 2006. *SST 2.0: user manual*. Universidad Antonio de Nebrija. *Available at http://www.nebrija.es/∼jmaestro/esa/docs/SST-UserManual2-0.pdf*.

Seshia SA, Li W, Mitra S. 2007. Verification-guided soft error resilience. In: *Proceedings of the design automation and test in Europe (DATE)*. Piscataway: IEEE DOI 10.1109/DATE.2007.364501.

Silicon Graphics. 2000. *Standard template library, SGI [Online]*. Milpitas: Silicon Graphics. *Available at http://www.sgi. com/tech/stl*.

Thornton M, Drechsler R, Gunther W. 2000. A method for approximate equivalence checking. In: *Proceedings of the 30th IEEE international symposium on multiple-valued logic. Portland, OR*. Piscataway: IEEE, 447–452.