Verification-guided Voter Minimization in Triple-Modular Redundant Circuits

Dmitry Burlyaev Pascal Fradet Alain Girault

Inria; Univ. Grenoble Alpes
{first}.{last}@inria.fr

***Abstract*—We present a formal approach to minimize the number of voters in triple-modular redundant sequential circuits. Our technique actually works on a single copy of the circuit and considers a user-defined fault model (under the form "at most 1 bit-flip every k clock cycles"). Verification-based voter minimization guarantees that the resulting circuit (i) is fault tolerant to the soft-errors defined by the fault model and (ii) is functionally equivalent to the initial one. Our approach operates at the logic level and takes into account the input and output interface specifications of the circuit. Its implementation makes use of graph traversal algorithms, fixed-point iterations, and BDDs. Experimental results on the ITC'99 benchmark suite indicate that our method significantly decreases the number of inserted voters which entails a hardware reduction of up to 55% and a clock frequency increase of up to 35% compared to full TMR. We address scalability issues arising from formal verification with approximations and assess their efficiency and precision.**

I. Introduction

Circuit tolerance towards soft (non-destructive, non-permanent) errors is an important research topic. As technology shrinks to feature sizes below $0.25\mu m$, the risk of soft errors increases [1]. Triple-Modular Redundancy (TMR) proposed by von Neumann [2] remains the most popular fault tolerance technique in Field-Programmable Gate Arrays (FPGAs) to mask soft-errors. Yet, manual introduction of TMR [3] into a circuit design is often a tedious and error-prone process. Hence, automatic introduction of TMR is essential for fault tolerant FPGA designs.

In a triplicated sequential circuit, adding voters at the primary outputs is not sufficient in general. Indeed, an error may remain in a memory cell long enough till other errors occur. If another error occurs in a different copy of the circuit, then the final vote may produce an incorrect output. Voter insertion after each memory cell is sufficient to prevent errors from remaining in cells. However, it greatly increases hardware overhead and the critical path, which directly influences the circuit performance. In most cases, introducing a voter per cell is excessive. But, to the best of our knowledge, there is no tool dedicated to voter minimization in TMR that guarantees fault-tolerance according to a user-defined fault model. The main existing research trends in TMR are probabilistic-based.

Our objective is to propose an *automatic*, *optimized*, and *certified* transformation process for TMR on digital circuits.

This transformation process should insert as few voters as possible, while guaranteeing to mask the considered errors. In this paper, we focus on the optimization aspect of the automatic transformation.

We consider circuits described at the gate level (*i.e.,* netlists of AND, OR, NOT gates plus flip-flops (FFs) – also called memory cells). This level has two main advantages:

- gate level netlists can be described by an elementary language, which simplifies correctness proofs;
- it is easier to prevent synthesis tools to optimize (undo) our transformation at this late stage;

Since the main contributors to Soft-Error Rate (SER) are the FFs [4], we focus on errors caused by Single-Event Upsets (SEUs) (*i.e.,* bit-flips in FFs). We consider fault models of the form "at most one bit-flip within K cycles" denoted by $SEU(1, K)$.

The proposed voter-minimization methodology is based on a static analysis that checks whether an error in a single copy of the TMR circuit may remain after K cycles. If not, protecting the primary outputs with voters is sufficient to mask the error. If, for instance, the circuit is a pipeline without feedback loops, then any bit-flip will propagate to the outputs and will thus disappear before K cycles (assuming that all paths in the circuit are shorter than K). If the state of the circuit is still erroneous after K cycles (in the form of an incorrect value stored in one of its memory cells), there is a potential error accumulation since, according to the $SEU(1, K)$ model, another bit-flip may occur in another copy of the circuit. In this case, a voter is needed to prevent error accumulation.

Our static analysis consists of four steps. The first step, described in Sec. II, is purely syntactic and finds all loops in the circuit. Error accumulation can be prevented by keeping enough voters to cut all loops.

In many cases, a circuit resets (overwrites) some memory cells, which may mask errors. Detecting such cases allows further useless voters to be removed. This second step is performed by a semantic analysis (Sec. III) taking into account the logic of the circuit.

Circuits are also often supposed to be used in a specific context. For instance, a circuit specification may assume that a `start` signal occurs every x cycles and outputs are only read y cycles after each `start`. When such assumptions exist, taking them into account makes the semantic analysis more effective. These third and fourth steps are presented in Sec. IV and Sec. V for input and output specifications respectively.

Our analysis has been implemented based on graph algorithms and fixed point iterations using Binary Decision Diagram (BDD). We have tested several (safe) approximations and trade-offs between cost and precision. The implementation and experiments are presented in Sec. VI. Related work on TMR and voter insertion strategies are reviewed in Sec. VII. We summarize our contributions and sketch a few extensions in Sec. VIII.

II. Syntactic Analysis

We consider a triplicated circuit with voters but we actually work on a *single copy* of the circuit. The insertion or removal of voters is represented as the effect it would have on that single copy in the TMR circuit. We model a sequential circuit C as a directed graph G_C where each vertex represents a FF (memory cell or latch) and an edge $x \to y$ exists whenever there is at least one combinational path between the two FFs x and y in C. An error in a cell x may propagate, in the next clock cycle, to all cells connected to x by an edge in this graph. Note that this is an over-approximation since the error may actually be masked by some logical operation.

Under the fault model $SEU(1, K)$, error accumulation is the situation where an error remains in the circuit K clock cycles after the SEU that caused it. Indeed, any circuit C without feedback loop will return, after an SEU, to a correct state before K clock cycles, provided that K is larger than the maximal length of the simple paths in G_C (paths without repeating vertices). In environments with high levels of ionizing radiations (*e.g.,* space, particle accelerators), K is bigger than 10^{10} [5]. So, even if our approach can deal with any K, we can assume that K is larger than the max length of all simple paths in G_C. It follows that error accumulation can only be caused by cycles in G_C, which must therefore be cut by removing vertices. Removing a vertex in G_C amounts to protecting the corresponding memory cells with a voter in the triplicated circuit.

The best solution to cut all cycles in G_C is to find the Minimum Vertex Feedback Set (MVFS), *i.e.,* the smallest set of vertices whose removal leaves G_C without cycles. This standard graph problem is NP-hard [6] but there exist good polynomial time approximations [7]. The exact algorithm was efficient enough to be used in all our experiments.

A voter after each cell belonging to the MVFS prevents error accumulation[1]. This simple graph-based analysis is very effective with some classes of circuits. In particular, it is sufficient to remove all internal voters in pipelined architectures such as logarithm units and floating-point multipliers (see Table 3).

However, this approach is insufficient for many circuits due to the extensive use of loops in circuit synthesis from Mealy machine representation. In such circuits, most cells are in self-loops (e.g. D-type flip-flops with Enable input). This entails many voters if the syntactic analysis is used alone. However, if the circuit functionality is taken into account, we can discover that such memory cells may not lead to erroneous outputs. Detecting such cases requires to analyze the logic (semantics) of the circuit. We address this issue in the following section.

III. Semantic Analysis

The semantic analysis first computes the Reachable State Set (RSS) of the circuit with a voter inserted after each memory cell in the MVFS. Then, for each cell $m \in$ MVFS, it checks whether its voter is necessary: (i) the voter is removed and all possible errors (modeled by bit-flips of each cell in each state of RSS) are considered; (ii) if such an error leads to error accumulation, then the voter is needed and kept.

Correct and erroneous values are represented by the four-value logic domain D_1:

$$D_1 = \{0, 1, \overline{0}, \overline{1}\}$$

where $\overline{0}$ and $\overline{1}$ represent erroneous 0 and 1, respectively. The truth tables of standard operations in this four-value logic are given in Table 1. Note that AND and OR gates can mask errors: for instance, $\overline{0} \vee 1 = 1$ or $\overline{0} \wedge \overline{1} = 0$. The **err** function models bit-flips. The **vot** function models the effect of a voter on a single copy of the circuit, i.e., it corrects an error, $\overline{1}$ becoming 0 and vice-versa $\overline{0}$ becoming 1.

TABLE 1
Operators for 4-value logic domain D_1

operands	**0**	**1**	$\overline{\mathbf{1}}$	$\overline{\mathbf{0}}$
0	0	0	0	0
1	1	1	$\overline{1}$	$\overline{0}$
$\overline{\mathbf{1}}$	$\overline{1}$	1	$\overline{1}$	0
$\overline{\mathbf{0}}$	$\overline{0}$	1	1	$\overline{0}$

	NOT	**err**	**vot**
0	1	$\overline{1}$	0
1	0	$\overline{0}$	1
$\overline{\mathbf{1}}$	$\overline{0}$	0	0
$\overline{\mathbf{0}}$	$\overline{1}$	1	1

A sequential synchronous circuit with M memory cells and I primary inputs is formalized as a discrete-time transition system with the transition relation $\delta : \{0,1\}^M \times \{0,1\}^I \longmapsto \{0,1\}^M$. We abuse the notation and use M (resp. I) to denote both the number and the set of memory cells (resp. inputs) of the circuit. The state of a circuit is just the values of its cells and the initial state s_0 is obtained after the circuit reset. We write $\Delta(S)$ for the function returning the set of states obtained from the set S after one clock cycle. Formally

$$\Delta(S) = \{s' \mid \exists i.\ \exists s \in S.\ \delta(s, i) = s'\}$$

Δ applies the transition function δ to all states of its argument set and all possible inputs. The set of reachable states RSS is defined by the following iteration:

$$S_0 = \{s_0\} \qquad\qquad S_{i+1} = S_i \cup \Delta(S_i) \tag{1}$$

Starting from the initial state, we compute the set of reachable states by accumulating states obtained by applying iteratively Δ. The set of possible states being finite, the iteration reaches a fixed point representing the RSS denoted [2] by $\{s_0\}^*_\Delta$.

[1]For small Ks, other voters should be kept to cut paths greater than K.

[2]We will use this notation with other initial states and transition functions

The second phase is to check whether the suppression of voters may lead to an error accumulation under the fault-model $SEU(1, K)$. Let δ_V be the transition relation of a circuit equipped with a voter after each cell in a given set V, and let Δ_V be its extension to sets. δ_V is defined as:

$$\begin{array}{lrll} \multicolumn{4}{l}{\delta_V((m_1, \ldots, m_{_M}), i) = \delta((m'_1, \ldots, m'_{_M}), i)} \\ \text{where } m'_i & = & \mathbf{vot}(m_i) & \text{if } m_i \in V \\ & = & m_i & \text{otherwise} \end{array}$$

This checking process can be expressed as the algorithm in Fig. 1. It starts with the circuit equipped with a voter after

```
$V := MVFS;\ RSS := \{s_0\}^*_\Delta$
forall  $m \in MVFS$
    $V := V \backslash \{m\};$
    $S := \Delta^K_V(\bigcup_{m_i \in M} RSS[m_i \leftarrow \mathbf{err}(m_i)]$
    if  $ErrAcc(S)$ then $V := V \cup \{m\}$
```

Fig. 1. Semantic Analysis – Main Loop

each cell in the MVFS. For each such cell m, it checks whether its voter suppression entails error accumulation. A bit-flip is introduced in all possible cells and states of RSS ($\bigcup_{m_i \in M} RSS[m_i \leftarrow err(m_i)]$). The transition function corresponding to the circuit with the current set of voters (V) is applied K times (Δ^K_V). The set of states obtained shows error accumulation if there exists an erroneous cell in at least one state of this set, which we capture with the predicate *ErrAcc*:

$$ErrAcc(S) \Leftrightarrow \exists s \in S.\ \exists m \in s.\ m = \overline{0} \vee m = \overline{1}$$

If the set S does not show error accumulation, the voter is useless and can be left out. Otherwise the voter is re-introduced.

In practice, Δ is applied the small number of times dictated by the circuit functionality and available simulation time. It is always safe to stop the computation before reaching K; the only drawback is to infer an error accumulation when there is none. Furthermore, the iteration stops

- if the current set of states is errorless, then there cannot be error accumulation (no error can reappear);
- or, if the erroneous current set is the same as the previous one, a fixed point is reached and there is an error accumulation.

The order in which the cells in MVFS are analyzed may influence the number of voters introduced. We use heuristics to choose a satisfying order but we do not describe it here for lack of space.

This method is precise but costly since it considers all possible inputs. In general, keeping track of the relations between indeterminate inputs is not very useful. Fortunately, our technique can be used as it is with other logic domains. There are several domains that retain enough precision and allow larger circuits to be analyzed. The 4-value logic domain D_2 decreases the state explosion that occurs with D_1:

$$D_2 = \{0, 1, \mathrm{U}, \overline{\mathrm{U}}\}$$

The abstract value U represents a correct value (either 0 or 1) and $\overline{\mathrm{U}}$ represents any (possibly erroneous) value (*i.e.,* 0, 1, $\overline{0}$ or $\overline{1}$). A vector of n inputs is represented as a unique vector (U, . . . , U) in D_2 whereas 2^n vectors had to be considered in D_1. The truth tables of standard operations in D_2 are given

TABLE 2
OPERATORS FOR 4-VALUE LOGIC DOMAIN D_2

operands	**0**	**1**	**U**	**$\overline{\mathrm{U}}$**
0	0	0	0	0
1	1	1	U	$\overline{\mathrm{U}}$
U	U	1	U	$\overline{\mathrm{U}}$
$\overline{\mathrm{U}}$	$\overline{\mathrm{U}}$	1	$\overline{\mathrm{U}}$	$\overline{\mathrm{U}}$

	NOT	**err**	**vot**
0	1	$\overline{\mathrm{U}}$	0
1	0	$\overline{\mathrm{U}}$	1
U	U	$\overline{\mathrm{U}}$	U
$\overline{\mathrm{U}}$	$\overline{\mathrm{U}}$	$\overline{\mathrm{U}}$	U

in Table 2. In contrast with D_1, a gate with two erroneous values cannot produce a correct one. Logical masking of errors can only occur with two operations: $0 \wedge \overline{\mathrm{U}}$ and $1 \vee \overline{\mathrm{U}}$. This is sufficient to take into account masking performed by explicit signals (*e.g.,* resets).

Typical examples where the semantic analysis is effective are circuits that use D-type flip-flips with an `enable` input driven by a Finite State Machines (FSM) encoded in the circuit. The syntactic approach would keep a voter for each such cell (they are in self-loops). The semantic analysis can detect that such cells are regularly overwritten by fresh inputs. For example, the resource arbiter in Sec. VI is such a circuit. After initialization, its finite state machine forces many cells to be overwritten with fresh values every other cycle. The semantic analysis is able to show that those cells, although in self loops, do not need to be protected by voters.

IV. Inputs Specification

Circuits are often designed to be used in a specific context where some input signals must occur at definite timings. Taking into account assumptions about the context may make the semantical analysis much more precise, in particular, when the logical masking of corrupted cells depends on specific inputs (*e.g.,* a `start` control signal). Our approach is to translate these specifications into a new interface circuit feeding the original circuit with the specified inputs. The analysis of the previous section can be applied to that new combined circuit. As a consequence, error accumulation is checked as before but under the constraints specified by the interface. The only small adjustment needed in Fig. 1 is to make sure that errors are introduced only in the cells of the original circuit and not in the cells of the interface circuit.

We use ω-regular expressions to specify circuit interfaces. An ω-regular expression specifies constraints using vectors of $\{0, 1, \star\}$, which replace primary inputs by 0, 1, or leave them unchanged ($\star$ being the wild card). Consider, for instance, a circuit with two primary inputs $[i_1, i_2]$, then the expression $([1, 0]+[0, 1]).[\star, \star]^\omega$ specifies that the circuit first reads either $i_1 = 0$ and $i_2 = 1$ or $i_1 = 0$ and $i_2 = 1$ and proceeds with no further constraints.

In general, specifications need non-determinism to describe not fully specified or non-deterministic context. So, the aforementioned ω-regular expression can also be seen as a Nondeterministic Büchi Automaton (NBA) that reads inputs and replace them by 0, 1, or leave them unchanged ($\star$). The previous regular expression can be represented as the two-state NBA of Fig. 2 (a): in the first state, it reads inputs and returns either the outputs $[1, 0]$ or $[0, 1]$ (regardless of the inputs). Then, the automaton goes (and stays) in the second state where inputs are read and produced as outputs.

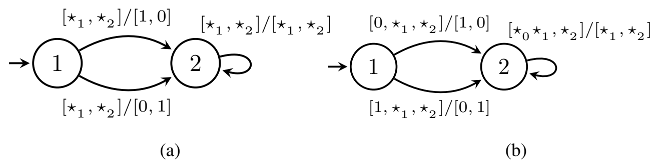
$[\star_1, \star_2]/[1, 0]$
$[\star_1, \star_2]/[\star_1, \star_2]$
1
2
$[\star_1, \star_2]/[0, 1]$
(a)
$[0, \star_1, \star_2]/[1, 0]$
$[\star_0 \star_1, \star_2]/[\star_1, \star_2]$
1
2
$[1, \star_1, \star_2]/[0, 1]$
(b)

Fig. 2. Input interface as a NBA (a) and its deterministic version (b)

To generate a circuit, we first convert the corresponding NBA into a deterministic automaton as follows. Each nondeterministic edge is made deterministic using new inputs (sometimes referred as oracles). If a vertex has n nondeterministic outgoing edges, adding $log_2(n)$ new inputs is sufficient. For example, the specification $([1, 0] + [0, 1]).[\star, \star]^\omega$ can be made deterministic by adding a single additional input i. The automaton (see Fig. 2 (b)) now reads three inputs: if i is 0 (resp. 1) it produces $[1, 0]$ (resp. $[0, 1]$). The resulting deterministic automaton is then translated into an interface circuit using standard logic synthesis techniques [8, p.118]. If the original circuit has I inputs, the resulting interface circuit will have $I + a$ (a new inputs to make it deterministic) inputs and I outputs. It is then plugged by connecting its outputs to the inputs of the circuit to be analyzed.

A typical example where input specification is useful is the circuit $b08$ of Sec. VI. Such a circuit has a `start` input signal and 8-bit data input. Its input interface specification can be expressed as the following ω-regular expression:

$$([1, \star, \star, \star, \star, \star, \star, \star, \star].[0, \star, \star, \star, \star, \star, \star, \star, \star]^{17})^\omega \tag{2}$$

A `start` signal is first raised and the input data is read. For the next 17 cycles data is processed and `start` kept to 0. This process is repeated over and over. Since `start` is raised every 18 clock cycles, the internal data registers are rewritten periodically with new data and can keep erroneous data only until the next `start` signal. The circuit has also an internal FSM which can be corrupted but the periodic `start` ensures that it returns to its initial state. Consequently, error accumulation is impossible (for large enough K) and no voters (except implicit voters at primary outputs) need to be inserted.

V. Outputs Specification

Consider another example, similar to the previous one, with 2 inputs, 1 output and where some waiting can occur before raising `start` signal. Formally, the input interface would be

$$([0, \star]^*.[1, \star].[0, \star]^{17})^\omega \tag{3}$$

This interface does not guarantee that `start` will be raised before K clock cycles. Since the analysis must consider the case where *start* is not raised, it may detect error accumulation even though *start* would ensure logical masking. However, if the primary outputs are not read before some useful computation triggered by the `start` signal completes, a better analysis can be performed.

We specify the output interface by adding to each vector of the input interface a vector of $\{0, 1\}$ indicating whether the corresponding outputs are read (1) or not read (0). For instance, the output interface of the previous example, where the single bit output is read only after `start` is raised, can be specified as

$$(([0, \star] : [0])^*.([1, \star] : [1]).([0, \star] : [1])^{17})^\omega \tag{4}$$

It states that the output is not read ($[0]$) until the `start` signal is raised. Then, the output is read ($[1]$) during 18 cycles.

The interface automaton is made deterministic as before. The corresponding interface circuit will additionally produce 0 or 1 signals to filter the outputs of the original circuit. Each such signal is connected using a AND gate to its corresponding primary output.

The property to check must now be refined to allow error accumulation as long as no error propagates to the (filtered) primary outputs. When an error occurs, it is allowed to propagate to outputs or voters before K clock cycles since no additional bit flip can occur during that time. However, the analysis must ensure that no error can propagate to outputs or voters after K cycles. This ensures that a second error, which can now occur in another redundant module of the TMR circuit, will only propagate alone to voters or outputs, and hence will be corrected.

When an error accumulation is detected in S (see Fig. 1), the fixed point (of the same iteration as Equation (1) but starting with S) $S^*_{\Delta_X}$ is computed. It represents all reachable states after a single error and K clock cycles. The analysis checks, for all states of that set, that no error propagates to the outputs or to the voters. This check is iterated for a second error and so on until a global fixed point is reached. Formally, the analysis computes the fixed point of the following iteration:

$$\begin{array}{lcl} E_0 & = & S^*_{\Delta_X} \\ E_{i+1} & = & E_i \cup (\Delta^K(\bigcup_{m_i \in M} E_i[m_i \leftarrow \mathbf{err}(m_i)])^*_{\Delta_X} \end{array}$$

At each bit-flip, we apply the transition function K times (in practice a safe and much lower bound) and compute a local fixed point E_i that represents the set of reachable states after i bit-flips. The global fixed point, written E^*, represents all reachable states after an arbitrary number of bit-flips separated from each other by at least K clock cycles.

To detect error propagation at the outputs, each primary output is represented by a new memory cell. Assuming that out (resp. vot) is a predicate telling whether a cell is an output (resp. a voter), checking error propagation is defined as:

$$\begin{array}{rl} \mathit{ErrProp}(E^*) \;\Leftrightarrow & \exists s \in S.\ \exists m \in s.\ (out(m)\ \vee\ vot(m)) \\ & \wedge\ (m = \overline{0}\ \vee\ m = \overline{1}) \end{array}$$

This check is added in the semantic analysis (Fig. 1) by performing the computation of E^* in the main loop (Fig. 1)

TABLE 3
VOTER MINIMIZATION AT ANALYSIS STEPS

	Circuit	FFs	Syn.	Sem. $D_1\backslash D_2$	Sem.Inp. $D_1\backslash D_2$	Sem.Out. $D_1\backslash D_2$
Data Flow I.	Pipe.FP.Mult.8x8 [11]	121	0	0\0	0\0	0\0
	Pipe.log.unit [11]	41	0	0\0	0\0	0\0
	Sh./A.Mult.8x8 [13]	28	28	19\19	19\19	8\8
	ITC'99 [12](subset)					
Control Flow Intensive	$b01$ Flows Compar.	5	3	3\3	3\3	3\3
	$b02$ BCD recogn.	4	3	2\3	2\3	2\3
	$b03$ Resourc.arbiter	30	29	17\29	17\29	17\29
	$b06$ Interrupt Hand.	9	3	3\3	3\3	3\3
	$b08$ Inclus.detect.	21	21	21\21	0\21	0\21
	$b09$ Serial Convert.	28	21	20\20	20\20	20\20

and replacing the conditional by:

$$\textbf{if } \mathit{ErrAcc}(S) \ \wedge \mathit{ErrProp}(E^*) \textbf{ then } \ldots$$

Output interfaces are especially useful for circuits whose outputs are not read before some input signal is raised and some computation is completed. For instance, the shift/add multiplier (see Sec VI) waits for a `start` signal. During that time, errors may accumulate in internal registers and propagate to the outputs, which are not read. When `start` occurs, fresh input data is read and written to internal registers (which are thus reset). The outputs are read only after the multiplication is completed and a `done` signal is raised.

VI. EXPERIMENTAL RESULTS

The presented voter minimization technique has been implemented using a BDD library. Transition systems and set of states are expressed by BDD formulas [9]. The introduced four-value logic domains (D_1 and D_2) are encoded by a pair of bits and the associated operators (Tables 1 and 2) are expressed as logic formulae over those pairs. We used the fault-model $SEU(1, K)$ with $K = 100$, which allows K cycles/transitions to be computed effectively (Δ^K). The obtained results are *a fortiori* valid for any $K \geq 100$.

BDDs proved to be quite efficient to express the data structures and the processing required by our technique. We made use of Rudell's sifting reordering [10] while building and applying the transition function. It allowed the semantic analysis of circuits up to 100 memory cells on a standard PC (Intel Core i5-2430M/2Gb-DDR3). We did not put much efforts in the optimization but believe that there remain much opportunities for improvement.

The algorithm has been applied to common arithmetic units taken from the *OpenCores* project [11] and to the *ITC'99* benchmark suite [12]. Table 3 summarizes the results on those circuits. The column FFs shows the total number of memory cells in the original circuit, while other columns show the number of voters in the TMR circuit after the syntactic and semantic steps (without, with input, with input/output interfaces). In each case, we give the results obtained with the D_1 and D_2 four-value domains.

The syntactic step eliminates all voters in circuits with a pipelined architecture such as adders, multipliers, or logarith-

TABLE 4
FREQUENCY AND AREA GAIN OF OPTIMIZED *vs* FULL TMR

	TMR circuit	voters	MHz	gain	hw	gain
Data Flow I.	Pipel.FP.Mult.8x8	121	60.5		2338	
	Optimized	0	71.0	17.4%	1831	21.7%
	Pipel.log.un.	41	128.3		693	
	Optimized	0	184.1	43.5%	447	35.5%
	Shift/Add.Mult.8x8	28	106.0		537	
	Optimized	8	108.0	1.9%	408	24.0%
Control Flow Intensive	$b01$ Flows Compar.	5	162.6		126	
	Optimized	3	162.6	0%	114	9.5%
	$b02$ BCD recogn.	4	181.9		69	
	Optimized	2	206.6	13.6%	60	13.1%
	$b03$ Resourc.arbiter	30	81.6		594	
	Optimized	17	109.0	33.6%	576	3.0%
	$b06$ Interrupt Hand.	9	144.8		168	
	Optimized	3	144.8	0%	134	20.2%
	$b08$ Inclus.detect.	21	115.4		484	
	Optimized	0	142.4	23.4%	216	55.4%
	$b09$ Serial Convert.	28	89.4		584	
	Optimized	20	95.0	6.3%	565	3.3%

mic units. With rolling pipelined architectures (*e.g.*, shift/add multiplier), a control part may require voter protection.

In general, control intensive circuits require a protection of their FSMs. Almost all memory cells of the serial flow comparator ($b01$) or the serial-to-serial converter ($b09$) have to be protected. Nevertheless, our analysis is capable of suppressing a significant amount of voters in many control intensive circuits.

The logic domain D_2 is, most of the time, precise enough. However, correcting a bit-flip in D_2 (*e.g.*, $0 \rightarrow \overline{\text{U}} \rightarrow \text{U}$) looses information. In some circuits, like $b03$ and $b08$, substantial logical error masking is performed by an FSM and the analysis fails to detect it.

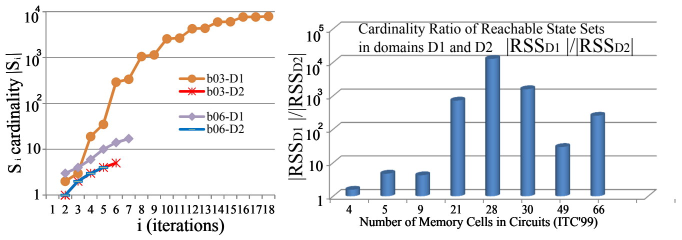

Fig. 3. Logic Domain Comparison: State Space Size

The scalability of logic domains D_1 and D_2 has also been compared. Fig. 3 presents the growth of S_i during the computation of the RSS (see Section III) for the $b03$ and $b06$ circuits. The fixed point is reached with less iterations in D_2, and the number of states growths exponentially for D_1 versus linearly for D_2. The same behavior is observed in all circuits of Table 3. The bar graph shows the ratio of the size of RSS in D_1 to the corresponding size in D_2. The RSSs in D_1 are several orders larger than the corresponding ones in D_2. The most computation demanding step of the whole analysis is checking error propagation (see Sec. V). A prohibiting growth of BDD structures representing the set of states E_i has been observed with D_1 for circuits of around 30 memory cells. The logic domain D_2 allows the analysis (with input and output interfaces) of much larger circuits (~ 100 cells).

In order to evaluate the benefits of our analysis, TMR has been applied to the benchmarks with the minimized set of voters. The final circuits have been synthesized with *Synplify Pro* with no optimization applied (Resource Sharing, FSM Optimization, etc.). As a case study, we have chosen Flash-based ProASIC3 FPGA as a synthesis target. Its configuration memory is immune to soft-errors [14] and data memory is protected with voters. Table 4 compares the size and maximum frequency of the circuit with full TMR (*i.e.,* voters after each FF) versus TMR with the optimized number of voters. The gains are presented in terms of the required FPGA hardware Core Cells (*hw* column) and maximum synthesizable frequency (*MHz* column). The gain in the maximum frequency depends on the location of the removed voters (in the circuit critical path or not). The reduction in area directly depends on the number of suppressed voters (up to 55%).

VII. Related work

Research on voter insertion and Selective Triple-Modular Redundancy (STMR) mainly focuses on probabilistic approaches [15]–[17] without any guarantee that the final circuit meets a fault-model. Johnson and Wirthlin [15] show how selective voter insertion minimizes the negative timing impact of TMR. In [18], probabilities are used to apply TMR on selected portions of the circuit (STMR). In [17], STMR of combinational circuits specifies input interfaces using input signal probabilities. The main advantage of STMR over TMR is that the area of the STMR circuit is roughly two-thirds of the area of the TMR circuit. However, since the proposed methods are probabilistic, some errors may propagate to primary outputs. In our approach, the circuit is guaranteed to mask all possible errors of the fault model.

Other work uses model-checking to guarantee user-defined fault-tolerance properties [19], [20]. While these studies do not address voter minimization, their formal approaches of fault-tolerance are related to our work.

VIII. Conclusion

We proposed a logic-level verification-guided approach to minimize the number of voters in TMR circuits that guarantees a user-defined fault-model to be masked. The approach is based on reachable state set computations and input/output interface specifications. In order to avoid analyzing the triplicated circuit, we introduced two four-value logic domains, which allow us to perform the analysis on a single copy of the circuit. Our analysis shows that some voters are useless and can be safely removed from the TMR application. We have used as case studies several arithmetic circuits as well as the benchmark suite *ITC'99*. They show that our technique allows not only a significant reduction in the amount of hardware resources (up to 35% for data flow intensive circuits and up to 55% for control flow intensive ones), but also a significant increase in the clock rate, compared to the full TMR method that inserts a voter after each memory cell.

We demonstrated that the choice of the logic domain influences the scalability of the analysis, while keeping enough precision for the analysed circuits.

For space concerns, several extensions have not been presented. In particular, we have considered another logic domain (combining 0, 1, $\overline{0}$, $\overline{1}$, U, and $\overline{\text{U}}$) and the extension of our analysis to Single-Event Transient (SET) (which may cause multiple cell upsets). We have started the formal certification of the approach using the Coq proof assistant. Further research directions include the generalization of our approach to other fault-tolerance techniques such as time redundancy and making the analysis modular to allow its application on much larger circuits.

Acknowledgments

Thanks are due to Bertrand Jeannet for valuable discussions about BDDs and implementation issues.

References

[1] N. Cohen, T. S. Sriram, N. Leland, D. Moyer, S. Butler, and R. Flatley, "Soft error considerations for deep-submicron CMOS circuit applications," *IEEE Int. Elect. Devices Meeting Tech. Dig.*, pp. 315–319, 1999.
[2] J. von Neumann, "Probabilistic logic and the synthesis of reliable organisms from unreliable components," *Automata Studies, Princeton Univ. Press*, pp. 43–98, 1956.
[3] S. Habinc, "Functional triple modular redundancy FTMR," *European Space Agency Contract Report*, no. FPGA-003-01, December 2002.
[4] T. Heijmen, "Soft-error vulnerability of sub-100-nm flip-flops," *14th IEEE Int.On-Line Testing Symposium*, pp. 247–252, 2008.
[5] A. Bogorad *et al.*, "On-orbit error rates of RHBD SRAMs: Comparison of calculation techniques and space environmental models with observed performance," *IEEE Trans. on Nuclear Science*, pp. 2804–2806, 2011.
[6] R. Karp, "Reducibility among combinatorial problems," *Complexity of computer computations*, vol. 43, pp. 85–103, 1972.
[7] G. Even, J. S. Naor, B. Schieber, and M. Sudan, "Approximating minimum feedback sets and multi-cuts in directed graphs," in *Proc. 4th Int. Conf. on Int. Prog. and Combinatorial Opt.*, 1995, pp. 14–28.
[8] G. De Micheli, *Synthesis and Optimization of Digital Circuits*, 1st ed. McGraw-Hill Higher Education, 1994.
[9] E. M. Clarke, J. R. Burch, O. Grumberg, D. E. Long, and K. L. McMillan, "Mechanized reasoning and hardware design," 1992, ch. Automatic verification of sequential circuit designs, pp. 105–120.
[10] R. Rudell, "Dynamic variable ordering for ordered binary decision diagrams," in *Proc. of CAD-93*, 1993, pp. 42–47.
[11] *Open Source Hardware IPs: OpenCores project*, Michael Dunn- Logarithm Unit; Launchbird Design Systems, Inc.-Floating Point multiplier, http://opencores.org/.
[12] F. Corno, M. Reorda, and G. Squillero, "RT-level ITC'99 benchmarks and first ATPG results," *Design Test of Computers*, pp. 44–53, 2000.
[13] S. Kilts, *Advanced FPGA Design: Architecture, Implementation, and Optimization.* Wiley-IEEE Press, 2007.
[14] "Neutron-induced single event upset SEU," *Microsemi Corporation*, no. 55800021-0/8.11, August 2011.
[15] J. M. Johnson and M. J. Wirthlin, "Voter insertion algorithms for FPGA designs using triple modular redundancy," in *FPGA*, 2010, pp. 249–258.
[16] B. B. Alagoz, "Fault masking by probabilistic voting," *OncuBilim Algorithm And Systems Labs*, vol. 9, no. 1, 2009.
[17] P. Samudrala *et al.*, "Selective triple modular redundancy based single-event upset tolerant synthesis for FPGAs," *IEEE Transactions on Nuclear Science*, pp. 284 – 287, October 2004.
[18] O. Ruano, P. Reviriego, and J. Maestro, "Automatic insertion of selective TMR for SEU mitigation," *European Conference on Radiation and Its Effects on Components and Systems*, pp. 284 – 287, 2008.
[19] S. Seshia, W. Li, and S. Mitra, "Verification-guided soft error resilience," in *DATE '07*, 2007, pp. 1–6.
[20] S. Baarir, C. Braunstein *et al.*, "Complementary formal approaches for dependability analysis," in *IEEE Int.Symp. on Defect and Fault Tolerance in VLSI Systems*, 2009, pp. 331–339.