Stuck-At Fault Mitigation of Emerging Technologies based Switching Lattices

Lorena Anghel
Grenoble-Alpes University,
TIMA Laboratory, France
lorena.anghel@univ-grenoble-alpes.fr

Anna Bernasconi
Dipartimento di Informatica
Università di Pisa, Italy
anna.bernasconi@unipi.it

Valentina Ciriani
Dipartimento di Informatica
Università degli Studi di Milano, Italy
valentina.ciriani@unimi.it

Luca Frontini
INFN
Milano, Italy
luca.frontini@mi.infn.it

Gabriella Trucco
Dipartimento di Informatica
Università degli Studi di Milano, Italy
gabriella.trucco@unimi.it

Ioana Vatajelu
Grenoble-Alpes University
TIMA Laboratory, France
ioana.vatajelu@univ-grenoble-alpes.fr

***Abstract*—Switching lattices are two-dimensional arrays composed by two or four-terminals switches organized as a crossbar array. The idea of using regular two-dimensional arrays of switches for Boolean function implementation was proposed by Akers in 1972. Recently, with the advent of a variety of emerging nanoscale technologies, lattices have found a renewed interest. Emerging technologies allow more complex functions integration thanks to their smaller technology sizes and different advantageous properties such as zero leakage current, capability to retain their data when in power-off state, almost unlimited endurance, to name just a few appealing features. Thanks to them, the implementation of new computing paradigms combining memory and logic altogether becomes possible. However, emerging technologies show a non-negligible defect ratio and important sensitivity to mismatches and process and environment variations. The reliability challenges of adopting these technologies need to be investigated. In this paper, we analyze the resilience of switching lattices face to stuck-at-fault model (SAF). We first identify the critical switches thanks to an elaborated sensitivity methodology and extensive analysis of the lattice. Next, we propose several techniques to improve lattice resilience face to these types of faults, implemented after lattice logic optimization steps.**

***Index Terms*—Switching lattices, Stuck-At-Fault model, fault injection, defect avoidance, fault tolerance.**

I. Introduction

Recent years advancements in process scaling and 3D monolithic integration brought multiple possibilities for emerging devices to push further integration of integrated circuits. Nano-crossbars are among the most promising alternative solutions to continue the process scaling down [29]. They lead to programmable logic circuits physically implemented as nano-crossbar arrays that operate similarly to the conventional programmable logic arrays (PLAs), or as molecular switch crossbar arrays, or resistive crossbar logic [2], [15]. Due to simpler and cheaper manufacturing techniques, programmable nano-crossbars arrays present more regular and dense forms [9] being also among the best with respect to area and power efficiency [2]. A given arithmetic logic function computation is mapped on a matrix of cross points that can be made of two-terminals switches i.e diodes [16], or resistive/memristive elements [23] or FET transistors [24]. Four-terminals switches is also a great possibility particularly well adapted to dual logic functions implementations [1], [20].

The memrisitve based cross points represents probably the most prominent solutions explored by many research groups [17], adopted due to their potential to scale down to 5nm offering much higher integration densities and compatibility to classical CMOS process.In addition to that, memristive non volatile memory (NVM) technologies have other very appealing features, such as zero leakage currents, longer retention of their data in the power off state, normally-off/instant-on capabilities, almost unlimited endurance. The non-volatility of these devices offer possibilities for memory-intensive computing paradigms, enabling non Von-Neuman logic-in-memory paradigm [7]. This paradigm, allows logic or arithmetic operations to be directly processed in the memory. Due to their versatile circuit-level implementations, memristive crossbar arrays are considered as serious candidate for pattern recognition, classifications, decision-making tasks processing, inherent to neuromorphic applications. [12], [22]. This paradigm allows logic and arithmetic operations to be directly processed in the memory within the crossbar array (In Memory Computing), making them attractive from neuromorphic applications such as prediction, classification and decision-making

Funded by the European Union Horizon 2020 research and innovation programme under the Marie Skłodowska-Curie grant agreement No 691178.

978-1-7281-1756-0/19/$31.00 ©2019 IEEE

problems [12], [22]. Memristive devices can be programmed to store either two (binary) or more than two (analog) states, when multiple resistance states are used together. Several non-volatile, memristive technologies are considered for the In-Memory Computing Paradigm, including Spin Transfer Torque magnetic Random Access Memory (STT RAM), Resistive Random Access Memory (ReRAM), Phase Change Random Access memory (PCRAM) to name just a few, all of them being suitable to In-Memory Computing implementations.

The majority of the emerging technologies are still immature, prone to important defect densities, manufacturing variations and mismatches, being also sensitive to temperature and voltage variations. Spot defects, dust, assemblage faults, imperfections of the fabrication process, instabilities, variations and mismatches drastically affect the fabrication yield but can also affect the reliability of the circuit in the field. They induce parametric faults, or logic faults and can be classified into two categories: soft faults and hard faults [11], [26]. Soft faults are caused by different cycle-to-cycle or device-to-device variations during the fabrication, but they can also manifest in-field during read/write operations, or as retention faults where the content of a cell can be lost after some time [27]. Hard faults are provoked either by fabrication steps as mentioned before, or by spot defects, extreme parametric variations or by the forming process or the continuous stress. The failures will induce read and write failures on the memory cell, as well as stuck-at faults. One typical type of hard fault occurs when the resistance of a resistive memory cell will no longer change; this category includes stuck-at-0 (SA0) and stuck-at-1 (SA1) faults caused by fabrication techniques and limited endurance. In this case, the faulty device is stuck-at-high resistance or stuck-at-low resistance state. These situations occur with a significantly high probability. It is reported that 63% of a storage array based on memristor is fault free in a 4Mb resistive RAM, with about 10% of the cells being of Stuck-At type [8]. In [28] the authors showed that 10% of broken memristor cells will lead to substantial degradation of the accuracy and overall performances of a convolutional neural network implemented on this structure. This study has been performed on crossbar arrays with binary resistive devices, and the considered faults are uniformly distributed. In addition, as mentioned previously, variability of memristor resistances during write operations may also push the device in a hard Stuck-at fault. However, to the best of our knowledge very limited research has been dedicated so far to analyze fault models of resistive devices where multiple resistances are used. Therefore most research works considered stuck-at faults as predominant to perform yield analysis.

Since the fabrication technology of crossbar cells is sensitive to different process steps (i.e., forming), it is very difficult to prevent stuck-at faults during the fabrication process [11], [8]. Understanding the impact of the sensitivity to stuck-at faults on the mapping algorithm is a key step for future developments. Yield analysis of memristive based nanocrossbars for uniformly and clustered distributed defects have been performed in [21], [25], [18]. Also various testing methods have been proposed in [10], [14], [19], [18]. Testing provides a faulty cells map that can be used not only for identifying faulty devices but it also helps managing the programmability around defected cells through algorithm remapping [28]). This generally requires important area overheads, as designer has to take into account margins in terms of sufficient spares organized in columns, lines or blocks.

In this paper, our contributions are as follows:

1) The sensitivity of arbitrary logic function decomposition algorithm on a given size crossbar is analyzed face to SA0 and SA1. 2) We propose fault injection methods and tools for crossbar arrays targeting stuck-at fault (SA1 or SA0) model. The fault injection algorithm uses independent uniform distribution as reported in [28].The experimental fault injection simulation aims at validating the decomposition method with single fault injection at the time. 3) The prior sensitivity analysis helps identifying critical switches. Further to that we propose mitigation factors to strengthen the mapping algorithm while keeping the crossbar array area minimal. This paper is an extension of a first contribution of the same authors published in IEEE Latin American Test Symposium in 2019 [4]. The innovative aspects here are related to new properties of synthesis algorithms for lattices, and a new algorithm to increase the fault tolerance level of lattices mapped on a defective crossbars. This is proved by new evaluation results. The paper is organized as follows. Section II explain the logic function synthesis method on crossbar switching arrays. Section III discuss the fault model and the sensitivity analysis method. Section IV and V discuss methods for mitigation and finally Section VII presents sensitivity results.

II. Switching Lattices and Synthesis Methods

Multi-terminals switching lattices are typically used as higher level models of switching nano-crossbar arrays. In fact, the circuit is represented by a single lattice composed by a two-dimensional array of two or four-terminal switches. Each of the terminals of a given switch links to its neighbours of the crossbar cell, so that these are either all connected (when the switch is ON), or all disconnected (when the switch is OFF).A Boolean function can be implemented by a lattice by ensuring proper connectivity across it, such as:

- in the case of the four-terminal switches, each terminals of the switch is controlled by a literal;

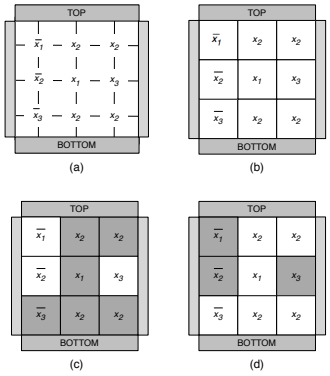

Fig. 1. A four terminal switching network implementing the function $f = \overline{x}_1\overline{x}_2\overline{x}_3 + x_1x_2 + x_2x_3$ (a); its corresponding lattice form (b); the lattice evaluated on the assignments 1,1,0 (c) and 0, 0, 1 (d), with grey and white squares representing ON and OFF switches, respectively.

- if the literal takes the value 1, the corresponding switch is connected to its four neighbours, otherwise it is considered as non connected;
- the function output yields 1 (output = 1) if and only if there exists a connected path between two opposing edges of the lattice, i.e., the top and the bottom edges;
- input assignments that leave the edges unconnected correspond to an invalid output (output = 0).

For instance, the network of switches shown in Figure 1 (a) corresponds to the lattice form depicted in Figure 1 (b), which implements the function $f = \overline{x}_1\overline{x}_2\overline{x}_3 + x_1x_2 + x_2x_3$. If we assign the values 1, 1, 0 to the variables x_1, x_2, x_3, respectively, we obtain paths of gray square connecting the top and the bottom edges of the lattices (Figure 1 (c)), on this assignment f output is 1. On the contrary, the assignment $x_1 = 0, x_2 = 0, x_3 = 1$, on which f evaluates to 0, does not create any path from the top to the bottom edge (Figure 1 (d)).

The synthesis objective on a lattice consists in finding an assignment of literals to switch terminals in order to implement a given logic function, the lattice being of the minimal size. Within this abstraction level, the size of a lattice is measured in terms of the number of switches in the lattice.

A switching lattice can similarly be equipped with left edge to right edge connectivity, so that a single lattice can implement two different functions. This fact is explained in [3] where the authors propose a synthesis method for switching lattices simultaneously implementing a function f according to the connectivity between the top and the bottom plates, and its dual function f^D according to the connectivity between the left and the right plates. Recall that the dual of a Boolean function f depending on n binary variables is the function f^D such that $f(x_1, x_2, \ldots, x_n) = \overline{f^D}(\overline{x}_1, \overline{x}_2, \ldots, \overline{x}_n)$. This method produces lattices with a size that grows linearly with the number of products in an non-redundant sum of product

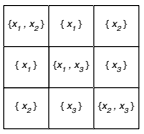

Fig. 2. A lattice for the function $f = x_1x_2 + x_1x_3 + x_2x_3$, with multiple choices on the diagonal cells.

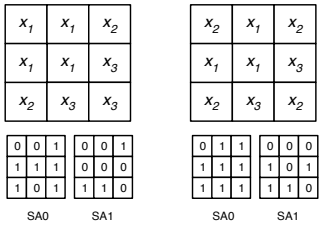

Fig. 3. Two lattices for $f = x_1x_2 + x_1x_3 + x_2x_3$ (see Figure 2 for the multiple choice lattice), with different sensitivity to SA0 and SA1 defects.

(SOP) representation of f, and consists of the following three steps:

1) find an non-redundant, or a minimal, SOP representation for f and f^D: $SOP(f) = p_1 + p_2 + \cdots + p_s$ and $SOP(f^D) = q_1 + q_2 + \cdots + q_r$;
2) assign each product p_j $(1 \leq j \leq s)$ of $SOP(f)$ to a column and each product q_i $(1 \leq i \leq r)$ of $SOP(f^D)$ to a row;
3) for all $1 \leq i \leq r$ and all $1 \leq j \leq s$, assign to the switch on the lattice site (i, j) one literal which is shared by q_i and p_j (the fact that f and f^D are dual guarantees that such a shared literal exists for all i and j).

Note that, we can have a couple of products q_i and p_j such that the intersection of their literals is a set of cardinality greater than one. For building the corresponding lattice, the algorithm imposes to choose randomly one of the common literals. In this case, we denote the corresponding cell (i, j) as a *cell with multiple choice*. Moreover, note that, in Step 2 of the synthesis algorithm, the assignments of products to rows and columns is random. For example, consider the function f in ISOP form $f = x_1x_2 + x_1x_3 + x_2x_3$ and its dual $f^D = x_1x_2 + x_1x_3 + x_2x_3$. The lattice containing the cells with multiple choice is depicted in Figure 2. At the top of Figure 3, we have two possible lattices derived from the former by choosing a literal in the cells with multiple choices.

A second approach to the synthesis of minimal-sized lattices is proposed in [13], where the authors transform the synthesis problem into a satisfiability problem in quantified Boolean logic and solve it using a quantified Boolean formula solver.

III. Single Stuck-At-Faults in Lattices

The well-known Stuck-at Fault (SAF) model is the most used logic model covering a large number of defects, and is

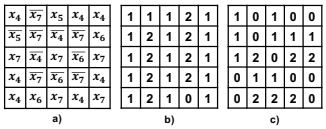

Fig. 4. a) Lattice design for the example function f and its sensitivity map for b) SA0 and c) SA1.

classically used by the semiconductor industry for many years. In CMOS and emerging technologies, the SA model assumes that a defect causes a basic cell input or output to be fixed to either 0 or 1. Thus, all defects covered by this logic fault model can be further detected by stuck-at-fault tests generated by structural, functional, or cell-aware ATPG. In a lattice, a SAF can be similarly modeled as a fixed value (0 or 1) of the faulty cell (i.e., a four-terminal switch) of the lattice. As our objective is to evaluate the sensitivity of a lattice with respect to SA0 and SA1 faults. We propose some metrics to quantify for a given function, the average number of defective outputs of this function. In this section we briefly summarize the methodology described in [20] for the fault injection, which we exploit for the sensitivity analysis of our approach. The fault injection is performed uniformly by substituting a single cell at a time with an always stuck-at-1 (SA1) or stack-at-0 (SA0) cell. The fault injection procedure is then repeated for each cell of the lattice . The simulation algorithm generates all 2^n possible inputs and for each input $x_1, \ldots, x_n$ the output is compared with the correct one (i.e., $f(x_1, \ldots, x_n)$.

Let r and s be the number of rows and columns, respectively, in a lattice. Let E^0_{ij} (resp., E^1_{ij}), with $1 \leq i \leq r$, $1 \leq j \leq s$, be the number of defective outputs when a SA0 (resp., SA1) affects a cell (i, j) of the given lattice. E^0_{ij} (resp., E^1_{ij}) is equal to 0 when, for any possible input, the output of the function mapped on the lattice is not affected by the SAF in the cell (i, j). In this case, the cell (i, j) is considered *robust* w.r.t. SA0 (resp., SA1). Let R^0 (resp., R^1) be the total number of robust cells w.r.t. SA0 (resp., SA1) in the lattice. Let $E^0 = \sum_{i=1}^{i=r} \sum_{j=1}^{j=s} E^0_{ij}$ (resp., $E^1 = \sum_{i=1}^{i=r} \sum_{j=1}^{j=s} E^1_{ij}$) be the total number of defective outputs with respect to SA0 (resp. SA1), considering the entire lattice.

Consider, for example, the function $f = x_4\overline{x}_5x_7 + \overline{x}_4x_6\overline{x}_7 + \overline{x}_4x_5\overline{x}_6x_7 + x_4\overline{x}_6\overline{x}_7 + x_4x_6x_7$ represented by the lattice in Figure 4 (a) (derived with the method in [3]). Figures 4 (b) and 4 (c) show the sensitivity map containing the value E^0_{ij} and E^1_{ij} in each cell. In order to evaluate the sensitivity of a lattice to SA0 and SA1 defects, we propose a metric that computes the average number of defective outputs: *Sensitivity of lattice* is expressed as the ratio between the total number of inputs that are able to propagates a SA0 or a SA1 into a faulty output and the total number of possible inputs. In the case of SA0, $S^0_L = E^0/(2^n(r \times s))$, and for SA1, $S^1_L = E^1/(2^n(r \times s))$.

IV. Analysis of lattices synthesized with the Altun-Riedel method

In this paragraph, some characteristics of the switching lattices obtained with the Altun-Riedel synthesis method [3] are discussed. These characteristics will be further exploited to enhance their fault tolerance.

First of all, we note that the Altun-Riedel method allows definition of many equivalent lattices for the given function f. Indeed, in the second step of the procedure (see Section II) each product in an non-redundant SOP for f is assigned to a column, and each product in an non-redundant SOP for the dual f^D is assigned to a row, without any specific rule for these assignments. As a consequence, any permutation of the products in $SOP(f)$ and in $SOP(f^D)$ gives rise to a correct, and possibly different, lattice for f. Moreover, once each pair of products (one from $SOP(f)$ and one from $SOP(f^D)$) has been assigned to a lattice cell, the controlling literal is selected choosing arbitrarily one of the literals shared by both products. Thus, we could have multiple choices for the controlling literals in some lattice cells.

Taking into account these degrees of freedom, we now evaluate the number of potentially different lattices produced by this synthesis procedure. Suppose that $SOP(f)$ contains s products, and $SOP(f^D)$ contains r products. The lattice for f function has the size of $r \times s$. Let us denote by $S(i, j)$ the subset of literals shared by the products assigned to the cell (i, j), and by $s_{i,j}$ the cardinality of this set. We have

Proposition 1: The number N_f of lattices for f produced by the Altun-Riedel method is given by $N_f = r!s! \prod_{\substack{1 \leq i \leq r \\ 1 \leq j \leq s}} s_{i,j}$.

Proof. Immediately follows since there are $r!$ ways to assign the products of $SOP(f^D)$ to the rows of the lattice, $s!$ ways to assign the products of $SOP(f)$ to the columns, and $s_{i,j}$ ways to select the controlling literals of each cell (i, j). ■

Observe that N_f can be exponential in the lattice size:

Corollary 1: Let f depend on n binary variables. Then $N_f = O(r!s!n^{rs})$.

Proof. Easy to follow as $s_{i,j} \leq n$. ■

Thus, the Altun-Riedel method provides many equivalent lattices for the same specified function f, all of them of $r \times s$ size. These lattices may exhibit a different SAF sensitivity. Consider, for example, the lattice for $f = x_1x_2 + x_1x_3 + x_2x_3$ depicted in Figure 2, with cells with multiple choice on the diagonal. Starting from this lattice we can build up to $288 = 3!3!8$ lattices by permuting rows and columns and by choosing the controlling literal for the diagonal cells. For instance, by simply making different choices on the diagonal cells, we can obtain the two lattices presented in Figure 3. We can see that they exhibit different sensitivities to SA0 and SA1 defects: $S^0_L = 1/12$ and $S^1_L = 1/24$ for the first lattice, and $S^0_L = 1/9$ and $S^1_L = 1/18$ for the second one. In particular, the first

lattice contains more robust cells, probably as a consequence of the many adjacent cells with the same controlling literal that might help contain the effect of a faulty cell. Therefore, instead of picking a random permutation of the products in the starting assignment of SOPs, and selecting arbitrarily the controlling literal for all cells with multiple choice, one should exploit the degrees of freedom offered by the Altun-Riedel method to detect, among the N_f different lattices, the most resilient one. This issue will be discussed in the next Section V.

Another interesting property of lattices synthesized with the Altun-Riedel method is that the function implemented by the lattice does not change after the insertion of a duplicate of a column or of a row, as proved in the following proposition.

Proposition 2: Let L be a lattice for a function f obtained with the Altun-Riedel method. The lattice L' derived inserting in L a duplicate of a column or of a row, implements the same function f.

Proof. The thesis is a direct consequence of the synthesis procedure, as adding a copy of a column means adding a copy of a product in the SOP for f, while adding a copy of a row means adding a copy of a product in the SOP for f^D, and this does not change the functions f and f^D. More precisely, we can observe that in the proof of correctness of the Altun-Riedel method, it is crucial that all products in $SOP(f)$ and $SOP(f^D)$ correspond to prime implicants, but the proof holds even if we replicate one or more products in the SOP expressions. Indeed, in this case the two functions f and f^D are still implemented as subsets of all top-to-bottom and left-to-right paths of the new lattice L', and this implies that L' correctly computes f and f^D (for more details, see Theorem 1 in [3]). ■

As a matter of fact, we exploit this property to enhance the single stack-at fault tolerance of a lattice, as further discussed in Section VI.

Finally, observe that the possibility of permuting and duplicating rows and columns is not guaranteed in lattices synthesized with other strategies. Consider for instance the 3×3 lattice L implementing the function $f = x_1x_2\overline{x}_3 + x_1\overline{x}_2x_3 + \overline{x}_1x_2x_3$ (see also Figure 5 (a)). This lattice has not been synthesised with the Altun-Riedel method, which would have instead produced a lattice of size 4×3. In fact, permuting columns of L and inserting in L a duplicate of a column, we can derive the two lattices in Figure 5 (b) and Figure 5 (c), respectively. However, we can observe that these lattices do not implement the function f, as they contain an valid path for the through $x_1x_2x_3$.

V. Construction and Characterization of Resilient Lattices

Let us consider a Boolean function f synthsized with Altun-Riedel method [3] . From the demonstration shown in Section IV, the synthesis algorithm can generate different possible lattices for this function f, which are exponential in number. The main aim of this section is to study efficient strategies to select the lattice that is less sensitive to cell SA0 or SA1 faults.

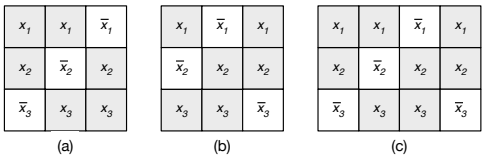

Fig. 5. A lattice L for $f = x_1x_2\overline{x}_3 + x_1\overline{x}_2x_3 + \overline{x}_1x_2x_3$ (a); a lattice obtained from L by a column permutation (b); a lattice obtained inserting in L a duplicate of the first column (c). All lattices are evaluated on the assignment $x_1 = 1, x_2 = 1, x_3 = 1$.

Let's call two cells in a lattice as being in the first-order neighborhood or *adjacent* if they are in the same column and in two adjacent rows or in the same row and in two adjacent columns. Consider the two lattices shown in Figure 6 both obtained by applying Altun-Riedel method to the function $f = x_1 + x_2x_4x_5 + x_3x_4x_5$. Let us assume a SA0 affects the first cell on the top-left (depicted in gray in the lattices). While the lattice on the left leads to the computation of an erroneous output, equivalent to a different function, i.e., $f' = x_1+x_3x_4x_5$, the lattice on the right computes the correct function even in the presence of the SA0 fault. In addition, we can observe that the lattice on the right is derived from the first one by a simple permutation of columns. In particular, in the second lattice, two similar columns are adjacent. This example gives us the intuition that, in order to decrease the sensitivity to a defective cell, we should bring cells containing the same literal in the first-order neighborhood. In fact, the product that is not computed anymore by the faulty version of the first lattice (i.e., $x_2x_4x_5$), is computed by the second lattice using a connection path starting at the top of the second column going down and then on the left (i.e., path x_4, x_5, x_5, x_2). Note that this connection path is not possible in the lattice on the left since the two involved columns are not adjacent.

Motivated by this observation, in the next sections we will describe several methods that through different permutations bring those cells containing the same controlling literals in a first-order neighborhood. Let us first give a metric that simplify the description of the proposed technique.

Consider a lattice L where each cell contains a controlling literal. For each cell c in L, with the controlling literal ℓ, we define a_c the number of cells adjacent to c in L containing the same controlling literal ℓ. Let a_L be $\sum_{c\in L} a_c$. In order to maximize the number of adjacent cells containing the same literal, we must maximize a_L.

As already observed, the synthesis algorithm proposed by Altun-Riedel produces a lattice containing cells with multiple

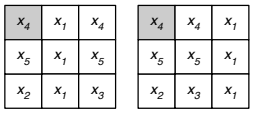

Fig. 6. Two equivalent lattices for the function $f = x_1 + x_2x_4x_5 + x_3x_4x_5$. While, in case of a SA0 in the first cell on top-left, the first lattice computes a different function, the second one still computes f.

choices (e.g., the lattice shown in Figure 2). The approaches we propose here in this paper are therefore based on three algorithms, each starting with a lattice (containing cells with multiple choices) produced by Altun-Riedel's algorithm with the following modifications:

- *PermuteColumns*: make a random choice for the cells with multiple choices and permute the columns in order to maximize the number of adjacent cells containing the same literal (i.e., a_L).
- *PermuteRows*: make a random choice for the cells with multiple choices and permute the rows of a given lattice in order to maximize the number of adjacent cells containing the same literal (i.e., a_L).
- *ChooseLiteral*: given a lattice containing cells with multiple choices, in each cell with multiple choices choose the literal that maximize the number of adjacent cells containing the same literal (i.e., a_L).

Note that the three algorithms return one of the N_f possible lattices produced by Altun-Riedel's algorithm. In other words, the proposed procedures make deterministic choices aiming at reducing the sensitivity to SA faults, instead of the random choices performed by the standard Altun-Riedel's algorithm.

In order to combine the three former approaches, we define a new metric that evaluate the neighborhood of two cells with multiple choices. Suppose that we have two adjacent cells with multiple choices (c_1 and c_2) containing the sets of literals L_1 and L_2, respectively. The *neighborhood of two adjacent cells* $n(c_1, c_2)$ is the cardinality of their intersections, i.e., $n(c_1, c_2) = |L_1 \cap L_2|$. The *neighborhood ($n(L)$) of a lattice* L containing cells with multiple choices is the sum of the neighborhoods of all the couples of adjacent cells contained in L.

We propose a strategy that starts with a lattice containing cells with multiple choices L produced by Altun-Riedel's algorithm. Then the following algorithm is applied:

Algorithm *ChooseAndPermute*:

1) permute rows and columns in order to maximize $n(L)$,
2) in each cell with multiple choices, choose the literal that maximizes the number of adjacent cells containing the same literal (i.e., a_L).

VI. Further Improvements of the Overall Lattice Resilience

In this section, we show how the resiliency to single stuck-at-faults (SA0 or SA1) of a switching lattice synthesized with the Altun-Riedel method can be further enhanced by duplicating selected columns or rows. First of all, we observe that if we duplicate a lattice column containing a non-robust cell c with respect to a SA0 (or a SA1) fault, then the new lattice becomes resilient to the fault in c, as it is proved by the following proposition.

Proposition 3: For a given function f, let L be a lattice obtained with the Altun-Riedel method, and let $c(i, j)$ be a defective cell with respect to a SA0 (of SA1). The lattice L' derived from L by adding a duplicate of the j-th column still computes f and is resilient to the SA0 in $c(i, j)$.

Proof. Let L' be the lattice derived from L by adding a duplicate of the j-th column (w.l.o.g, suppose that the duplicate column has been added as new last column on the right side of the initial lattice). Recall from Proposition 2 that the new lattice L', without stuck-at faults, still computes f and f^D.

Now, consider a SA0 (or SA1) fault in L' affecting cell $c(i, j)$, and observe that, since we are injecting the value 0 (or 1) in one cell of the lattice, the lattice affected by the fault is always correct on the off-set of f. In fact, a cell affected by a SA0 (or SA1) could imply that the lattice computes a faulty 0 (or 1) for an on-set input. Thus, to prove that L' is robust with respect to the SA0 (SA1) at $c(i, j)$ we must show that it is correct on the on-set of f.

Consider an on-set minterm w, and let p be a product that covers w in the SOP for f used to build the lattice. The column of L' associated to p contains, by construction, a literal from p in each cell, and forms a top-to-bottom accepting path for w. Suppose that w is covered by the product corresponding to the j-th defective column of L. Then, the duplication of this column guarantees the presence of a top-to-bottom accepting path in L' that can replace the path blocked by the SA0 (SA1) fault in $c(i, j)$. ■

This strategy can be applied for SA0 or to SA1, and consists in the duplication of a row containing a non-robust cell in order to make the lattice resilient to considered faults .

Proposition 4: Let L be a lattice for a function f obtained with the Altun-Riedel method, and let c be a defective cell with respect to SA1 fault. The lattice L' derived adding to L a duplicate of the i-th row still computes f and is resilient to a SA1 in $c(i, j)$.

Proof. Let L' be the lattice derived from L adding a duplicate of the i-th row (w.l.o.g, suppose that the duplicate row has been added at the bottom of the lattice). Recall from Proposition 2 that the new lattice L' still computes f and f^D.

Now, consider a SA1 fault in L', in cell $c(i, j)$. Since the value 1 is now injected into one cell of the lattice, the faulty

lattice is always correct on the on-set of f. Thus, to prove that L' is robust with respect to the SA1 in $c(i,j)$ we must show that it is correct on the off-set of f, since a cell SA1 could imply that the lattice computes a faulty 1 for an off-set input.

Let us consider a minterm w such that $f(w) = 0$, and a minterm w' obtained by complementing all literals in w. By the definition of the dual function, we know that $f^D(w') = 1$. Let p be a product that covers w' in the SOP for f^D used to build the lattice. The row of L' associated to p contains, by construction, a literal from p in each cell, and forms a left-to-right accepting path for w'. On the other hand, each cell on this row evaluates to 0 on input w (so that the non faulty lattice correctly outputs 0 on w). Suppose that w' is covered by the product of f^D corresponding to the i-th row of L. Then, the duplication of the i-th defective row guarantees the presence of a row where each cell evaluates to 1 on w' and to 0 on w, blocking any top-to-bottom accepting path possibly produced by the SA1 in $c(i,j)$. ■

Note that, in both cases (column or row duplication) the lattice implements correctly the original function f from top to bottom, even in presence of a SA fault in one of the cells. However, the function computed taking into account the left-to-right connectivity could be different from the dual of f. In fact, the defected cell $c(i,j)$ can still be critical for f^D and may induce changes on the output computed by L' with respect to the left-to-right connectivity.

An immediate consequence of Propositions 3 and 4 is that we can always obtain a lattice for a function f resilient to a single SA fault in any one of its cells, by duplicating all columns and all rows of a lattice for f derived with the Altun-Riedel method.

Corollary 2: A lattice L synthesized according to the Altun-Riedel method can be transformed into a resilient one to single stuck-at fault affecting any single cell by duplicating each row and each column.

Proof. Follows from Propositions 3 and 4. ■

Note however that this strategy if prohibitive as the size of the lattice becomes four times bigger.

A better approach in obtaining a resilient lattice to a single SA0 or SA1 is to pre-compute in advance a lattice with one spare row and one spare column, at the bottom and at the right side, initially filled with ones or zeroes, respectively, as depicted in Figure 7 (b) for the benchmark function *Newtag*. The area of the lattice increases from $r \times s$ to $(r+1) \times (s+1)$, which is better than $4(r \times s)$ obtained previously. The idea is to exploit the additional column and row to duplicate a row, or a column, containing a defective cell. To this aim, it is necessary to connect a multiplexer, whose inputs are all the original input literals, to each spare cell in order to map the desired literal to the switch in the cell. The multiplexer can be realized using $2n$ single lattice cells that have the top connected to the corresponding variable, the output connected to the spare cell and a control signal that controls the multiplexer cell, as shown in Figure 7 (e). At the beginning of the test procedure, the spare column will be filled with zeroes and the spare row with ones; if a defect is detected, then the spare cells are used to duplicate the column or the row containing the faulty cell, as shown in Figure 7 (c) and (d).

Taking into account all previous observations, we therefore propose the following strategy to mitigate the sensitivity of a lattice to SA faults in a more cost effective way.

- First, we synthesize the lattice with the Altun-Riedel synthesis method, which provides lattices less sensitive to SA faults, as experimentally verified (see [20]). As a matter of fact, the more compact the lattice is, the higher the output function sensitivity to SA0 or SA1 faults.
- Second, we apply the Algorithm *ChooseAndPermute* described in Section V as a further processing step after logic optimization, in order to improve the resiliency of the original lattice.
- For a given logic function, if a SA0 or SA1 fault affects a robust cell, as identified by the fault injection campaign, the lattice computes the correct output, and we do not need any further action of applying fault tolerant strategy.
- Otherwise, if an injected SA0 fault is proven to be critical for the output value, the column containing that defective cell is duplicated by a spare column using multiplexers. Similarly, in case of a SA1, the row that contains the defective cell is duplicated by a spare row.

VII. Experimental Evaluation

This section reports experimental results on the fault sensitivity of switching lattices face to the single stuck-at-fault model (SA0 and SA1). Our goal is to determine strategies that allow us to obtain less sensitive to SA0 and SA1 faults implementations for any logic function. For this purpose, for a selected benchmark, for each circuit from the benchmark we consider six different lattices:

- L: initial generic lattice that implements the function, where no algorithms have been applied to maximize the number of adjacent cells containing the same literal (i.e., [3] method)
- L_r: lattice obtained by the *PermuteRows* algorithm;
- L_c: lattice obtained by the *PermuteColumns* algorithm;
- L_{rc}: lattice obtained by applying both *PermuteRows* and *PermuteColumns* algorithms in the same time;
- L_ℓ: lattice obtained by the *ChooseLiteral* algorithm.
- $L_{c\&p}$: lattice obtained by applying the *ChooseAndPermute* algorithm.

The permutation of rows and columns as well as the computation of the best combinations of rows and columns

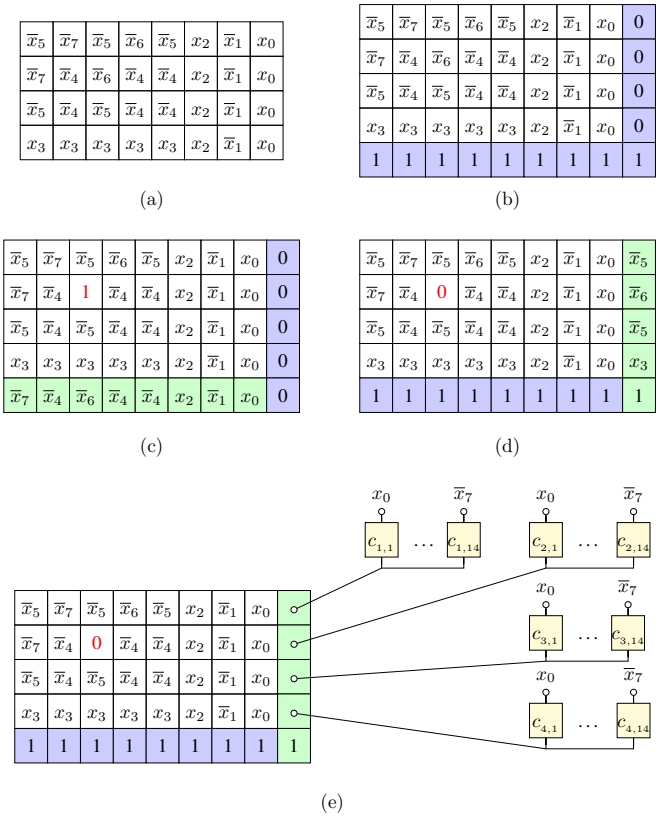

Fig. 7. (a) Lattice implementation for the benchmark *Newtag*, built from the SOPs $f = x_3\overline{x}_5\overline{x}_7 + x_3\overline{x}_4\overline{x}_7 + x_3\overline{x}_5\overline{x}_6 + x_3\overline{x}_4\overline{x}_6 + x_3\overline{x}_4\overline{x}_5 + x_2 + \overline{x}_1 + x_0$ and $f^D = x_0\overline{x}_1x_2\overline{x}_5\overline{x}_6\overline{x}_7 + x_0\overline{x}_1x_2\overline{x}_4\overline{x}_6\overline{x}_7 + x_0\overline{x}_1x_2\overline{x}_4\overline{x}_5 + x_0\overline{x}_1x_2x_3$; (b) a lattice for *Newtag* with a spare row and a spare column; (c) the lattice for *Newtag* resilient to the SA1 in $c(2,3)$ thanks to the use of the spare row; (d) the lattice for *Newtag* resilient to the SA0 in $c(2,3)$ thanks to the use of the spare column; (e) mapping of a column into the spare one, using multiplexers.

TABLE I

A COMPARISON OF S_L^0 AND S_C^0 BETWEEN LATTICES L, L_r, L_c, L_{rc}, L_ℓ, AND $L_{c\&p}$. THE BEST RESULTS ARE SHOWN IN BOLD FACE.

			L (initial lattice)		*L_r (PermuteRows)*		*L_c (PermuteColumns)*		*L_{rc} (PermuteRows, PermuteColumns)*		*L_ℓ (ChooseLiteral)*		*$L_{c\&p}$ (ChooseAndPermute)*	
	$r \times s$	n	S_L^0	S_L^1	S_L^0	S_L^1	S_L^0	S_L^1	S_L^0	S_L^1	S_L^0	S_L^1	S_L^0	S_L^1
al2(0)	3×5	7	**0.133**	0.013	0.136	**0.011**	**0.133**	0.013	0.136	**0.011**	**0.133**	0.013	0.136	**0.011**
al2(13)	6×5	7	0.007	0.047	**0.003**	**0.039**	0.005	0.050	**0.003**	**0.039**	0.007	0.047	**0.003**	**0.039**
alcom(2)	2×4	5	**0.016**	0.070	0.023	**0.063**	**0.016**	0.070	0.023	**0.063**	**0.016**	0.070	0.023	**0.063**
alu2(2)	11×10	8	0.012	0.004	0.005	0.002	0.013	0.004	0.006	0.002	0.012	0.004	**0.000**	**0.001**
b9(0)	9×10	7	0.020	0.009	0.019	0.007	0.014	0.010	0.014	0.007	0.002	0.007	**0.008**	**0.003**
b11(3)	3×6	6	0.043	0.038	0.033	**0.028**	0.038	0.039	**0.026**	**0.028**	0.043	0.038	**0.026**	**0.028**
bench(7)	4×6	6	0.284	**0.317**	**0.281**	0.320	0.283	**0.317**	0.282	0.320	0.284	0.319	0.289	0.322
clpl(0)	4×4	7	0.092	**0.025**	0.092	**0.025**	**0.087**	**0.025**	**0.087**	**0.025**	0.092	**0.025**	**0.087**	0.025
dc2(3)	12×14	7	0.003	0.005	0.003⋆	**0.002**⋆	**0.002**⋆	0.005⋆	0.003⋆	0.003⋆	0.003	0.005	**0.002**⋆	0.030⋆
fout(9)	10×12	6	0.030	0.021	0.021⋆	**0.005**⋆	0.025	**0.005**	**0.016**⋆	**0.005**⋆	0.019	0.006	0.024⋆	0.019⋆
in6(0)	2×7	8	0.034	0.021	0.036	0.020	0.034	0.021	0.036	0.020	**0.001**	**0.001**	0.051	0.071
luc(7)	4×7	6	0.040	0.009	0.032	**0.006**	0.036	0.009	0.031	**0.006**	0.040	0.009	**0.022**	**0.006**
luc(13)	9×10	6	0.012	0.006	0.011	**0.004**	0.010	0.006	0.008	**0.004**	0.015	**0.004**	**0.007**	0.005
newtag(0)	8×4	8	**0.032**	**0.004**	**0.032**	**0.004**	**0.032**	0.005	**0.032**	0.005	**0.032**	**0.004**	**0.032**	0.014
pope.rom(7)	15×11	6	0.014	0.004	0.013	**0.002**	0.010⋆	0.005⋆	0.009⋆	0.003⋆	0.009	0.003	**0.008**⋆	**0.002**⋆
rd53(2)	16×16	5	0.018	0.008	0.018⋆	0.004⋆	0.011⋆	0.015⋆	**0.008**⋆	0.004⋆	**0.008**	0.010	0.003⋆	**0.0036**⋆

TABLE II

COMPARISON OF THE LATTICES OBTAINED BY APPLYING THE PROPOSED METHODS WITH RESPECT TO THE LATTICES L OBTAINED WITH THE STANDARD ALTUN-RIEDEL METHOD.

	L_r (PermuteRows)		*L_r (PermuteColumns)*		*L_{rc} (PermuteRows, PermuteColumns)*		*L_ℓ (ChooseLiteral)*		*$L_{c\&p}$ (ChooseAndPermute)*	
	% more resilient lattices	average gain	% more resilient lattices	average gain	% more resilient lattices	average gain	% more resilient lattices	average gain	% more resilient lattices	average gain
SA0	42%	17%	44%	19%	52%	26%	24%	20%	68%	57%
SA1	56%	25%	12%	16%	55%	23%	17%	22%	58%	42%

has been done by using linear optimizer GLPK (GNU Linear Programming Kit). The simulation of GLPK on each input

case is stopped after 1 hour in case of an optimal solution is not found. If the simulation is stopped after 1 hour without obtaining the optimal solution, GLPK still produces a metric. This metric represents the percentage of the maximum relative gap between the value of the objective function for the best known integer feasible solution and the global bound for the exact integer optimum. In Table I, we mark these cases with the symbol '$\star$'.

For a given a benchmark - LGSynth93 [30], we computed the 6 lattices for each function and in each of them we performed error injection campaigns. During these campaigns, SA0 and SA1 faults have been uniformly injected in the lattices, a single fault at a time, and metrics described in Sect. III have been computed in order to evaluate the proposed strategies. The experiments have been run on a machine with two Intel Xeon E5-2683 for a total of 64 CPUs and 756 GByte of main memory, running Linux CentOS 7. The benchmarks functions are expressed in PLA form. A total of about 620 functions were considered, and each function has been implemented as a separate Boolean function. The software used for fault injection simulations is written in C++.

As the fault-injection simulations are preformed on all the possible inputs, which are exponential in number, we considered only benchmarks whose corresponding lattices have a number of variables smaller or equal to 8. Note that this limitation is due to the onerous procedure for the fault simulation, and it is not due to the proposed algorithms. In fact, the proposed algorithms can be applied to the entire benchmark set.

In Table I we report the sensitivity of lattices to SA0 and SA1 faults. Due to lack of space, the reported values in table I belong to a much smaller subset of the functions. The first column reports the name and the output number of the considered benchmark function; the second column reports the lattice dimension ($r \times s$) and the number of inputs n. The following columns report, by group of two, the results of the metric described in Sect III, for SA0 and SA1 respectively for each computed lattice.

Table II presents in a more compact way: 1) the percentages of lattices obtained with our methods, with higher resilience to SA faults with respect to the corresponding lattices synthesized with the standard Altun-Riedel method (*% more resilient lattices*); and 2) the percentages of average gain in resiliency (*average gain*). The computation of the percentages reported in Table II have been done on lattices with at least two columns and two rows.

First of all, we note that all proposed techniques allow to improve the resilience to SA-faults. Second, it is good to note that higher percentages are obtained with the algorithm *ChooseAndPermute* that exploits all the proposed mitigation approaches (i.e., *PermuteRows, PermuteColumns,* and *ChooseLiteral*). In the case of SA0, the percentages obtained by applying the *PermuteRows* and the *PermuteColumns* algorithms are comparable (42% and 44% respectively). On the other hand, for SA1 faults (last row of Table II), we can observe that the *PermuteRows* method allows to obtain higher resilience percentages (56%) with respect to *PermuteColumns* (12%). The column L_ℓ, referring to the *ChooseLiteral* method, shows the lowest resilient percentages, as expected; this is due to the low number of cells that present multiple literal choices. Therefore, this approach alone without row and column permutation is not interesting. Thus, the correct choice of the literal in multiple choice cells is a useful feature only when it is used together with row and column permutations. In fact, the algorithm *ChooseAndPermute* gives better results than the strategy that exploits just permutation of rows and columns (see lattices $L_{c\&p}$ and L_{rc} in Table II).

Therefore, the highest average resiliency gain is obtained when applying the *ChooseAndPermute* algorithm (57% for SA0 and 42% for SA1), and the lowest are obtained respectively with the *PermuteRows* method for SA0 (17%) and with the *PermuteColumns* method for SA1 (16%).

The evaluation of the technique presented in Section VI shows that *ChooseAndPermute* method guarantees, as expected, the complete lattice resilience to single SA1 or SA0, per lattice. Further to that, the resiliency improvement is independent of the lattice size.

Therefore, it could be useful, in further studies, to use this method for more complex arithmetic-logic functions used in massive parallel computation.

As a final consideration, we remark that the proposed methods are designed to improve the robustness of the overall lattice with respect to a single stuck-at-faults. In order to validate the proposed techniques, we needed to simulate the behaviour of the lattices considering all possible inputs. To deal with the high computational cost we have considered in this study single stuck-at fault only. Based on preliminary research work published by some authors of this paper [5], [6], we will extend the sensitivity evaluation to multiple, fist order neighbours clustered SAFs, and temporary faults (e.g. soft errors), for which an error detection and correction phase should be deployed in addition to the redundant fault avoidance remapping technique.

VIII. Conclusion

Emerging nanoscale technologies are very promising for circuit level implementation in cross-bar structures. However, due to their immature process, they can have non-negligible defect ratio and important variability. In this paper, we have proposed a method to analyze fault sensitivity of switching lattices under the single stuck-at-fault model (SAF). Algorithmic improvements of the fault resilience has been proposed, exploiting

different redundant schemes, such as literal selection, row or column permutations, or combination of both.

REFERENCES

[1] Akers, S.B.: A Rectangular Logic Array. IEEE Transactions on Computers **C-21**(8), 848–857 (1972). DOI 10.1109/TC.1972.5009040

[2] Alexandrescu, D., Altun, M., Anghel, L., Bernasconi, A., Ciriani, V., Frontini, L., Tahoori, M.: Logic synthesis and testing techniques for switching nano-crossbar arrays. Microprocessors and Microsystems **54**, 14–25 (2017). DOI 10.1016/j.micpro.2017.08.004

[3] Altun, M., Riedel, M.D.: Logic Synthesis for Switching Lattices. IEEE Transactions on Computers **61**(11), 1588–1600 (2012). DOI 10.1109/TC.2011.170

[4] Anghel, L., Bernaconi, A., Ciriani, V., Frontini, L., Trucco, G., Vatajelu, I.: Fault mitigation of swithcing lattices under stuck-at fault model. In: Proc.of IEEE Latin American Test Symposium, LATS 2019, Satiago de Chile, March 2019. IEEE (2019)

[5] Bernasconi, A., Ciriani, V., Frontini, L.: Testability of switching lattices in the stuck at fault model. In: proc. IFIP/IEEE International Conference on Very Large Scale Integration, VLSI-SoC 2018, Verona, Italy, October 8-10, 2018, pp. 213–218. IEEE (2018)

[6] Bernasconi, A., Ciriani, V., Frontini, L.: Testability of switching lattices in the cellular fault model. In: proc. 22nd Euromicro Conference on Digital System Design, DSD 2019, Kallithea, Greece, August 28-30, 2019, pp. 320–327. IEEE (2019)

[7] Borghetti, J., S Snider, G., J Kuekes, P., Yang, J.J., Stewart, D., Williams, S.: Memristive switches enable stateful logic operations via material implication. Nature **464**(7290), 873–876 (2010). DOI 10.1038/nature08940

[8] Chen, C., Shih, H., Wu, C., Lin, C., Chiu, P., Sheu, S., Chen, F.T.: Rram defect modeling and failure analysis based on march test and a novel squeeze-search scheme. IEEE Transactions on Computers **64**(1), 180–190 (2015)

[9] Chen, Y., Jung, G.Y., Ohlberg, D.A.A., Li, X., Stewart, D.R., Jeppesen, J.O., Nielsen, K.A., Stoddart, J.F., Williams, R.S.: Nanoscale molecular-switch crossbar circuits. Nanotechnology **14**, 462–468 (2003). DOI 10.1088/0957-4484/14/4/311

[10] Chen, Y., Li, J.: Fault modeling and testing of 1T1R memristor memories. In: Proc. IEEE 33rd VLSI Test Symposium (VTS), pp. 1–6 (2015)

[11] Degraeve, R., Fantini, A., Raghavan, N., Goux, L., Clima, S., Govoreanu, B., Belmonte, A., Linten, D., Jurczak, M.: Causes and consequences of the stochastic aspect of filamentary RRAM. Microelectronic Engineering **147**, 171 – 175 (2015)

[12] Deng, Y., Huang, P., Chen, B., Yang, X., Gao, B., Wang, J., Zeng, L., Du, G., Kang, J., yan Liu, X.: RRAM Crossbar Array With Cell Selection Device: A Device and Circuit Interaction Study. IEEE Transactions on Electron Devices **60**(2), 719–726 (2013)

[13] Gange, G., Søndergaard, H., Stuckey, P.J.: Synthesizing optimal switching lattices. ACM Transactions on Design Automation of Electronic Systems **20**(1) (2014). DOI 10.1145/2661632

[14] Hamdioui, S., Taouil, M., Haron, N.Z.: Testing open defects in memristor-based memories. IEEE Transactions on Computers **64**(1), 247–259 (2015)

[15] Haselman, M., Hauck, S.: The Future of Integrated Circuits: A Survey of Nanoelectronics. Proceedings of the IEEE **98**(1), 11–38 (2010)

[16] Huang, Y., Duan, X., Cui, Y., Lauhon, L.J., Kim, K.H., Lieber, C.M.: Logic gates and computation from assembled nanowire building blocks. Science **294**(5545), 1313–1317 (2001)

[17] ITRS: The International Technology Roadmap for Semiconductors. In: ITRS 2011 Edition (2011)

[18] Kang, W., al: Yield and reliability improvement techniques for emerging nonvolatile stt-mram. EEE J. Emerg. Sel. Topics Circuits Syst. **5 no.1**, 28–39 (2015)

[19] Kannan, S., Karri, R., Sinanoglu, O.: Sneak path testing and fault modeling for multilevel memristor-based memories. In: Proc. IEEE 31st International Conference on Computer Design (ICCD), pp. 215–220 (2013)

[20] Morgul, M.C., Tunali, O., Altun, M., Frontini, L., Ciriani, V., Vatajelu, E.I., Anghel, L., Moritz, C.A., Stan, M.R., Alexandrescu, D.: Integrated synthesis methodology for crossbar arrays. In: proc. IEEE/ACM International Symposium on Nanoscale Architectures (NANOARCH), pp. 1–7 (2018)

[21] Naeimi, H., DeHon, A.: A greedy algorithm for tolerating defective crosspoints in nanopla design. In: Proc. IEEE International Conference on Field-Programmable Technology, pp. 49–56 (2004)

[22] Ni, L., Huang, H., Liu, Z., Joshi, R.V., Yu, H.: Distributed in-memory computing on binary RRAM crossbar. ACM Journal on Emerging Technologies in Computing Systems **13**(3) (2017)

[23] Snider, G.: Computing with hysteretic resistor crossbars. Applied Physics A **80**(6), 1165–1172 (2005)

[24] Snider, G., Kuekes, P., Hogg, T., Williams, R.S.: Nanoelectronic architectures. Applied Physics A **80**(6), 1183–1195 (2005)

[25] Su, Y., Rao, W.: Defect-tolerant logic mapping on nanoscale crossbar architectures and yield analysis. In: Proc. 24th IEEE International Symposium on Defect and Fault Tolerance in VLSI Systems, pp. 322–330 (2009)

[26] Vatajelu, E.I., Prinetto, P., Taouil, M., Hamdioui, S.: Challenges and solutions in emerging memory testing. IEEE Transactions on Emerging Topics in Computing **7**(3), 493–506 (2019)

[27] Xia, L., Gu, P., Li, B., Tang, T., Yin, X., Huangfu, W., Yu, S., Cao, Y., Wang, Y., Yang, H.: Technological exploration of RRAM crossbar array for matrix-vector multiplication. Journal of Computer Science and Technology **31**(1), 3–19 (2016). DOI 10.1007/s11390-016-1608-8

[28] Xia, L., Liu, M., Ning, X., Chakrabarty, K., Wang, Y.: Fault-tolerant training with on-line fault detection for RRAM-based neural computing systems. In: Proc. 54th Design Automation Conference (DAC), pp. 1–6 (2017)

[29] Yan, H., Choe, H.S., Nam, S., Hu, Y., Das, S., Klemic, J.F., Ellenbogen, J.C., Lieber, C.M.: Programmable nanowire circuits for nanoprocessors. Nature **470**(7333), 240–244 (2011). DOI 10.1038/nature09749

[30] Yang, S.: Logic Synthesis and Optimization Benchmarks User Guide Version 3.0. User guide, Microelectronic Center (1991)