Write Sneak-Path Constraints Avoiding Disturbs in Memristor Crossbar Arrays

Yuval Cassuto[*], **Shahar Kvatinsky**[*], and **Eitan Yaakobi**[†]
[*]Electrical Engineering Department, Technion – Israel Institute of Technology
[†]Computer Science Department, Technion – Israel Institute of Technology
ycassuto@ee.technion.ac.il, shahar@ee.technion.ac.il, yaakobi@cs.technion.ac.il

***Abstract*—We study the problem of write disturbs due to write sneak paths in memristor crossbar arrays. A write sneak path is a bit configuration in the array that causes a write of one cell to undesirably flip the value of another cell. We study the configurations that cause such write sneak paths, and characterize them in terms of tight constraints to prevent them. We show that thanks to the flexibility to choose the write order, the resulting constraints are milder compared to known similar ones for read sneak paths. In addition, we derive the array constraints when parallel write is allowed in the rows or column only, and in both the rows and columns.**

I. Introduction

In recent years, different resistive memory technologies have emerged. Although these technologies include numerous different materials and physical mechanisms, they are all basically a resistor with a varying resistance and can be collectively considered as memristors [4], [5], [7]. Memristive technologies have raised high interest since they are non-volatile, and have superior density and endurance as compared to Flash memories. To achieve maximal density, memristors are organized in a crossbar array structure [8], [12], which is a two-dimensional grid such that on the intersection of every row and column there is only a memristor, as shown in Fig. 1. Since memristors are two-port devices, memristive crossbar memories suffer from interference between different memory cells both in read and write operations. Previously, we considered the read interference, namely the sneak path phenomenon [1]–[3]. Here we consider the write disturb phenomenon from an information theory perspective.

In memristive memories, each memristor is a memory cell and can either be in logical state 0 or 1, which depends on its resistance. Memristor in state 0 has high resistance R_{OFF}, and a state 1 memristor has low resistance R_{ON}. A *cell SET operation* is achieved by applying voltage V_{SET} that sets the memristor to logical state 1. The application of a SET operation involves a flow of current through the memristor in order to change its state. However, this current can build-up an undesired voltage in some other parts of the array, which may flip the content of unselected memory cells, thereby introducing a *write disturb* due to *sneak current paths* while programming the memristor [6], see Fig. 1.

Although it is possible to eliminate sneak paths by adding a CMOS transistor to each memory cell and isolate the cell from its neighbors, this approach significantly reduces density, and is therefore undesired. Other approaches to reduce sneak-path currents, such as *half selecting* part of the memory cells, are usually used [9], but not without keeping some sensitivity to write disturbs across the array. Previously, we studied a similar phenomenon occurring in read operations due to sneak paths [3]. In this paper, we explore the relevant conditions for write sneak paths, which are different than read sneak paths.

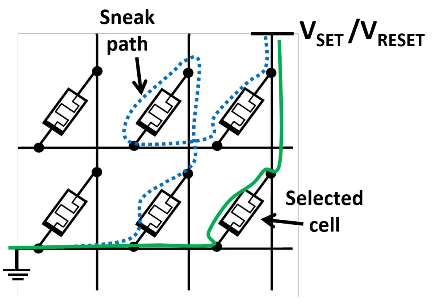

Fig. 1. Write sneak path.

II. Problem Definition: Understanding the Programming Constraints

To overcome or minimize the effect of sneak paths while writing, it is important to first understand the electrical conditions of the behavior that causes other memristors to change their state. When three memristors are connected in series, where two of them are at ON state (low resistance) and the third memristor is at OFF state (high resistance), the voltage upon the third memristor is approximately the applied voltage, and the voltage upon the other memristors is zero. In this case, the OFF memristor can change its state if the applied voltage has the magnitude of V_{SET} with the right polarity. The write disturb can occur both when V_{SET} and V_{RESET} are applied, but in this paper we focus on the more severe SET case. If voltage V_{SET} is applied on three consecutive memristors at states $(1,1,0)$ $(R_{ON}, R_{ON}, R_{OFF})$ or $(0,1,1)$ $(R_{OFF}, R_{ON}, R_{ON})$, then the memristor at logical value 0 changes its state to 1, that is $R_{OFF} \to R_{ON}$; see Fig. 2 (a-b). We introduce this error behavior formally.

Definition 1. *Given a binary array A of size $m \times n$, we say that a **SET write sneak path** occurs when programming the cell at position (i,j), if while applying to cell $a_{i,j}$ a change $0 \to 1$ (by voltage V_{SET}), there exist 2 positive integers $1 \leqslant r_1 \leqslant m$ and $1 \leqslant c_1 \leqslant n$, such that*

$$a_{i,c_1} = 0, a_{r_1,c_1} = a_{r_1,j} = 1,$$

or

$$a_{i,c_1} = a_{r_1,c_1} = 1, a_{r_1,j} = 0.$$

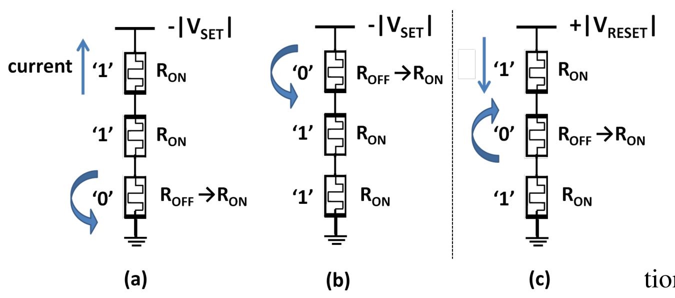

Fig. 2. (a-b) SET write sneak paths. (c) RESET write sneak path (not studied here).

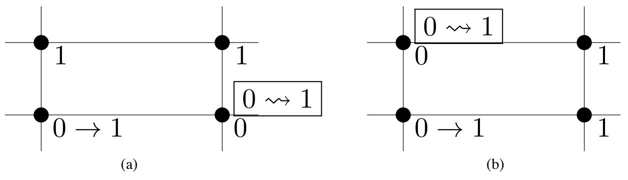

Fig. 3. Rectangles forming SET write sneak paths.

In this case the memristor in state 0 (in position (i, c_1) in the first case and (r_1, j) in the second case) changes $0 \to 1$ undesirably.

Definition 1 is illustrated in Fig. 3. This definition can be generalized as well for longer write sneak paths in an equivalent way to read sneak paths as explained in [3], however in this work we only consider sneak paths of length 3, which are the dominant sneak paths.

Since write sneak paths have an asymmetric behavior, depending on whether the cell is programmed to state 0 or 1, the order in which the memristors are programmed can affect the occurrence of write sneak paths. Assume that initially all the memristors are in state 0. Thus, all the programming transitions are $0 \to 1$, and only SET write sneak paths may take effect. In this case we only need to avoid the SET write sneak paths. Given a rectangle in the array that needs to be programmed to the target levels

$$a_{i_1,j_1} = 1, a_{i_1,j_2} = 0, a_{i_2,j_1} = 1, a_{i_2,j_2} = 1, \qquad (1)$$

(as depicted on Fig. 3a), it is clear that avoiding a SET write sneak path is possible by programming a_{i_2,j_1} (the top left cell on Fig. 3a) $0 \to 1$ at the last step. By that neither of the three $0 \to 1$ changes will result in SET write sneak path. It is also clear that an order where a_{i_2,j_1} is *not* last will result in a SET write sneak path. Trouble starts when different rectangles enforce conflicting orders to avoid sneak paths. To deal with such conflicts, we make the following definition.

Definition 2. *A binary array A contains a* ***SET conflict configuration*** *if there are indices i_1, i_2, i_3 and j_1, j_2, j_3 that make the following combinations*

$$\begin{array}{lcl} (a_{i_1,j_1}, a_{i_2,j_1}) = (1,1) & & (a_{i_1,j_1}, a_{i_1,j_2}) = (1,1) \\ (a_{i_1,j_2}, a_{i_2,j_2}) = (1,0) & \text{or} & (a_{i_2,j_1}, a_{i_2,j_2}) = (1,0) \\ (a_{i_1,j_3}, a_{i_2,j_3}) = (0,1) & & (a_{i_3,j_1}, a_{i_3,j_2}) = (0,1) \end{array}$$

An example of a SET conflict configuration is given in Fig. 4, which corresponds to the first option in Definition 2. Row-column transposition of the figure gives the second option in the definition.

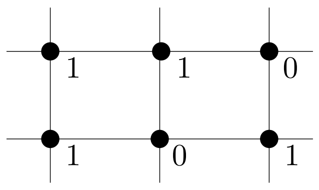

Fig. 4. SET conflict configuration.

To summarize this discussion, under this programming order, the patterns which we seek to avoid are the ones where there are two rows with the patterns $(1, 0), (1, 1), (0, 1)$ in some three columns on the two rows. A similar constraint is enforced to the columns. In order to analyze the properties of these arrays, we define the constraint on the memristors in the array.

Definition 3. *A binary array A contains two memristors in* ***conflict*** *on the j_1-th column, in positions $(i_1, j_1), (i_2, j_1)$ if there exist two other columns j_2, j_3 such that*

$$(a_{i_1,j_1}, a_{i_2,j_1}) = (1,1),$$
$$(a_{i_1,j_2}, a_{i_2,j_2}) = (1,0),$$
$$(a_{i_1,j_3}, a_{i_2,j_3}) = (0,1).$$

Memristors in conflict on a row are defined similarly. Lastly, we say that an array A is ***conflict-free*** *if it does not have two memristors in conflict (either in the row or column).*

III. Characterization of Conflict-Free Arrays

In this section we study the combinatorial properties and characterization of conflict-free arrays.

Let A be a binary $m \times n$ array. For $1 \leqslant i \leqslant m$, $1 \leqslant j \leqslant n$, let $R_i \subseteq [n], C_j \subseteq [m]$ be the locations set of the ones in this row, column, respectively, that is $R_i = \{k : \ a_{i,k} = 1\}$, and $C_j = \{\ell : \ a_{\ell,j} = 1\}$. We say that two rows are *in conflict* if none of the following three conditions hold: $R_{i_1} \subseteq R_{i_2}$, $R_{i_2} \subseteq R_{i_1}$, or $R_{i_2} \cap R_{i_1} = \emptyset$, and otherwise they are *not in conflict*. Columns in conflict and not in conflict are defined similarly. From the characterization of cells in conflict, it is implied that in order not to have cells in conflict, the 1 cell locations in any pair of rows (or columns) must be a subset of one another or must not overlap, that is, any two rows and columns are not in conflict. This property is proved in the next lemma.

Lemma 4. *An array A is conflict-free if and only if it does not have rows and columns in conflict.*

Proof: Assume in the contrary that the condition holds but the array A is not conflict-free. Assume without loss of generality that there are two memristors in conflict in a column. Then, according to Definition 3, there exist two different positive integers i_1, i_2 and three different positive integers j_1, j_2, j_3 such that

$$(a_{i_1,j_1}, a_{i_2,j_1}) = (1,1),$$
$$(a_{i_1,j_2}, a_{i_2,j_2}) = (1,0),$$
$$(a_{i_1,j_3}, a_{i_2,j_3}) = (0,1).$$

Then, we get that for the two rows i_1 and i_2: $j_1, j_2 \in R_{i_1}, j_3 \notin R_{i_1}$ and $j_1, j_3 \in R_{i_2}, j_2 \notin R_{i_2}$. Thus, none of the three conditions $R_{i_1} \subseteq R_{i_2}$, $R_{i_2} \subseteq R_{i_1}$, or $R_{i_2} \cap R_{i_1} = \emptyset$ hold.

To prove necessity, let us assume that there are two rows with indices $1 \leqslant i_1 < i_2 \leqslant m$, such that none of the three conditions holds: $R_{i_1} \subseteq R_{i_2}$, $R_{i_2} \subseteq R_{i_1}$, or $R_{i_2} \cap R_{i_1} = \emptyset$. Therefore, there exist j_1, j_2, j_3 such that $j_1 \in R_1 \setminus R_2$, $j_2 \in R_2 \setminus R_1$, and $j_3 \in R_1 \cap R_2$. Hence we get that $(a_{i_1,j_1}, a_{i_2,j_1}) = (1,1), (a_{i_1,j_2}, a_{i_2,j_2}) = (1,0), (a_{i_1,j_3}, a_{i_2,j_3}) = (0,1)$, and therefore the memristors in positions (i_1, j_1) and (i_2, j_1) are in conflict. ■

Combining the two conditions to have no rows and no columns in conflict, one can observe that they together imply the condition that

$$\begin{aligned} &\text{If } R_{i_2} \subseteq R_{i_1} \text{ and } R_{i_3} \subseteq R_{i_1}, \\ &\text{then } R_{i_2} \subseteq R_{i_3} \text{ or } R_{i_3} \subseteq R_{i_2}. \end{aligned} \tag{2}$$

That is, two sets contained by the same set must be in containment relation themselves (and cannot be disjoint). Next we show that arrays satisfying these constraints can be successfully programmed.

Theorem 5. *If an array A is conflict-free then it is possible to successfully program its cells.*

Proof: The programming order works as follows. Initially all cells are at logical state 0. Note that if the array A is conflict-free, then according to Lemma 4, no two rows or columns are in conflict.

Individual cells will be programmed row by row. That is, all cells in a particular row are programmed and only then proceeding to program the cells in another row. The programming order of any two rows, i_1 and i_2, is determined as follows:

1) If $R_{i_2} \cap R_{i_1} = \emptyset$ then either row can be programmed first.
2) If $R_{i_2} \subseteq R_{i_1}$, then row i_2 is programmed before row i_1.
3) If $R_{i_1} \subseteq R_{i_2}$, then row i_1 is programmed before row i_2.

According to this order we avoid any write sneak path that may affect two memristors in the same column. Lastly, within each row we program the ones starting from the column index that intersects with the fewest contained rows, and ending with the column index that intersects with the most. By that we resolve all the memristors in conflict in the programmed row.

■

On the other hand, we show that arrays which are not conflict-free prevent successful programming in case the initial logical state of all the cells is 0. These arrays cannot be successfully programmed under any programming order.

Theorem 6. *If an array A is not conflict-free then it is not possible to successfully program its cells, under any programming order.*

Proof: Examining the conflict configuration in Fig. 4, we see that the cells of the left-most column cannot both be programmed before any of the two other ones. Which means that one of the cells in the left column must be last. But clearly each choice of the last programmed cell will cause a sneak path to one of the two rectangles of form (1) existing in the configuration.

■

IV. Characterization of the Number of Conflict-Free Arrays

In this section we study the number of arrays which are conflict-free. Our main goal is to approximate the number of conflict-free arrays. We denote by $M(m,n)$ the number of conflict-free arrays of size $m \times n$.

In [3], [10], the number of arrays satisfying the *read sneak-path rectangle constraint* was studied. In the read sneak-path rectangle constraint it was required that every two rows and columns are either *identical* or the location sets of the ones are *disjoint*. Thus, the read sneak-path constraint is a stronger constraint than the conflict-free constraint, since there cannot be partial inclusion between any two rows or columns. We denote by $N(m,n)$ the number of $m \times n$ arrays satisfying the read sneak-path rectangle constraint. It was shown in [3], [10] that for m, n large enough the number of information bits which is possible to represent by these arrays behaves like $(m+n)\log(m+n)$, that is $\log N(m,n) \approx (m+n)\log(m+n)$. We seek to derive a similar analysis for the conflict-free constraint. Our main result here is the following connection between $M(m,n)$ and $N(m,n)$.

Theorem 7. *For all m, n,*

$$M(m,n) \leqslant N(m,n) \cdot (m+n)^{m+n}.$$

Proof: Given an $m \times n$ array A, recall that for $i \in [m]$ the set $R_i \subseteq [n]$ denotes the locations set of the ones in this row, and the set $C_j \subseteq [m]$ for $j \in [n]$ is defined similarly for the columns. We also refer here by a weight of a row, column, to the number of ones in this row, column, respectively. Assume for now that there are neither zero rows nor zero columns.

According to Lemma 4, an array is conflict-free if and only if it does not have rows and columns in conflict. By the definition of rows in conflict, we can define for every row $i \in [n]$, a row index $i_{\max}$ of a row with maximum weight which contains the set R_i as a subset and if there is more than one such row we choose the one with the minimum index. Assume there are ℓ different $i_{\max}$ indices, then the rows are partitioned into ℓ non-empty subsets $S_1, \ldots, S_\ell$ where for every set $S_i, i \in [\ell]$ there exists an index $i_{\max}$ such that for all $j \in S_i$, $R_j \subseteq R_{i_{\max}}$.

Since the columns are also not in conflict we could repeat the same process for the columns. However, if we continue with the partitions of the rows into the ℓ sets $S_1, \ldots, S_\ell$, then we only need to require that for every set $i \in [\ell]$ and $j_1, j_2 \in S_i$ it holds that $R_{j_1} \subseteq R_{j_2}$ or $R_{j_2} \subseteq R_{j_1}$.

The partition of the rows into ℓ non-empty subsets defines also a partition of the columns into ℓ non-empty subsets. Given a set of x rows which is matched with a set of y columns we study the number of options to position the ones in these x rows and y columns.

Proposition 8. *Assume a set of x rows is matched with a set of y columns. Then there are at most $(x+y)!$ options to place the locations of the ones in these x rows and y columns.*

Proof: Assume that the sets of x rows and y columns form an $x \times y$ array. According to the partition of the rows described above, there is at least one row with y ones, and for every two rows $i, j \in [x]$ it holds that $R_i \subseteq R_j$ or $R_j \subseteq R_i$. Hence we can form a permutation of the x rows and y columns such that the ones in each row are aligned to the left and the following holds:

$$R_1 \subseteq R_2 \subseteq \cdots \subseteq R_x = [y].$$

Let $r_1 = |R_1| > 0$ and for $2 \leqslant i \leqslant x$, $r_i = |R_i \setminus R_{i-1}| \geqslant 0$, then we have that $\sum_{i=1}^{x} r_i = y$. The number of solutions for this equation is given by

$$\binom{x+y-2}{x-1},$$

and since there are $x!$ options to permute the rows and $y!$ to permute the columns, the number of options to place the locations of the ones in the x rows and y columns is at most

$$\binom{x+y-2}{x-1} x! y! \leqslant (x+y)!.$$

■

Next, assume that the rows and columns are partitioned into ℓ sets and the sizes of the row sets are $x_1, \ldots, x_\ell$ and the sizes of the column sets are $y_1, \ldots, y_\ell$. Then, the number of arrays with this partition of rows and columns is at most

$$\begin{aligned}
&(x_1+y_1)! \cdot (x_2+y_2)! \cdots (x_\ell+y_\ell)! \\
\leqslant &(x_1+y_1)^{x_1+y_1} \cdot (x_2+y_2)^{x_2+y_2} \cdots (x_\ell+y_\ell)^{x_\ell+y_\ell} \\
\leqslant &(m+n)^{x_1+y_1} \cdot (m+n)^{x_2+y_2} \cdots (m+n)^{x_\ell+y_\ell} \\
= &(m+n)^{m+n}.
\end{aligned}$$

Lastly, we need to calculate the number of options to partition the rows and columns into some ℓ subsets. However, this value was already studied in [3] and is given by $N(m,n)$. Therefore, we conclude that

$$M(m,n) \leqslant N(m,n) \cdot (m+n)^{m+n}.$$

■

The asymptotic implications of Theorem 7 are that the log of $M(m,n)$ behaves at most like $2(m+n)\log(m+n)$, and as with the read sneak-path constraint [10], here too capacity vanishes to 0 as m and n tend to infinity and the ratio m/n approaches some positive number. Nevertheless, for fixed-dimension arrays the potential extra factor $(m+n)^{m+n}$ may lead to higher storage efficiency.

V. Parallel Programming

The programming order we discussed so far assumes the cells are programmed individually, one at a time. However, in practice, it is more common to program multiple cells in a row or column simultaneously [6], [8]. This parallel writing method eliminates sneak paths, because two memristors are no longer in conflict if programmed together; see Fig. 5.

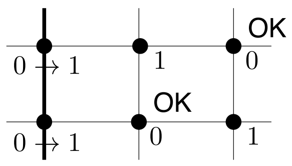

Fig. 5. Parallel programming in the columns.

We assume in this section that the cells are programmed in parallel, where their initial state is the logical value 0. When a row is chosen to be programmed, all the memristors in this row which need to store logical value 1 are programmed together by a SET operation. The same principle holds for parallel programming in the columns.

A. Parallel programming in the rows

The benefit of the parallel programming is that, if a full row is programmed in parallel, sneak paths can only affect memristors in other rows and not in the programmed row. Thus, if two memristors in the same row are in conflict, they will no longer cause an error since they are programmed together. However, two memristors in conflict in the same *column* will cause an error, since they cannot be programmed together. From symmetry, the same applies (reversed) when parallel programming is done in the columns.

We can thus prove the equivalent of Theorems 5 and 6.

Theorem 9. *An array can be successfully programmed in parallel in the rows if and only if it has no memristors in conflict in the columns.*

Proof: Let us first assume that the array has no memristors in conflict in the columns and we will show that it can be successfully programmed. Since the rows are programmed in parallel, we only need to determine their programming order. This order is identical to the one given in Theorem 5 since every two rows satisfy one of the three conditions we specified in the proof of Theorem 5.

To see the other direction, note that if two cells are in conflict in the columns then they will not be programmed together. Hence depending on the programming order, one of them will be programmed after the other and introduce a sneak-path error. ■

As before, the number of arrays that can be successfully programmed in parallel in the rows can be bounded. Let us denote this number by $P(m,n)$ and we have the following relation.

Theorem 10. *For all m, n,*

$$P(m,n) \leqslant M(m,n) \cdot n! \cdot m^n.$$

Proof: The proof strongly builds on the proof of Theorem 7, with some modifications. We call an array with no conflicts in the columns a P-array, and an array with no conflicts in the rows or columns a M-array. We first observe that M-arrays are also P-arrays, because having no conflicts in the rows/columns implies having no conflicts in the columns. Now we want to bound how many more P-arrays than M-arrays there are. In M-arrays we have the condition (2) on the rows that implies that within a $x \times y$ partition all rows are in containment relation. But in P-arrays this is no longer the case, and we may have that rows will satisfy $R_{i_2} \subseteq R_{i_1}$, $R_{i_3} \subseteq R_{i_1}$, and also $R_{i_2} \cap R_{i_3} = \emptyset$. In a P-array consider two row sets $\mathcal{R}_1$,$\mathcal{R}_2$ where in each $\mathcal{R}_i$ all rows are in containment relations, all rows in $\mathcal{R}_1$ are disjoint from all rows in $\mathcal{R}_2$, and there exists a row R_j that contains all rows of both $\mathcal{R}_1$ and $\mathcal{R}_2$. We can link this configuration to an M-array with two partitions corresponding to $\mathcal{R}_1$ and $\{\mathcal{R}_2, R'_j\}$, where R'_j is the intersection of R_j with the maximal row in $\mathcal{R}_2$. The P-array can be obtained from this M-array by *stretching* R'_j to cover the sets of $\mathcal{R}_1$ as well. Every P-array can be obtained by taking an M-array and stretching rows in it to cover multiple partitions. Define the number of partitions in an M-array to be T. In each $x \times y$ partition, between 0 and $x-1$ rows can be stretched to cover some subset of the other $T-1$ partitions. So if we are not careful we may count $\prod_{i=1}^{T}(x_i \cdot 2^{T-1})$ possible stretchings, which looks like a lot. But then we observe that the stretchings of different partitions are dependent, and need to induce an order between the partitions. So the number of stretchings is bounded by $T! \prod_{i=1}^{T} x_i$, the number of ways to order the T partitions times the number of rows to stretch in each partition. Bounding x_i by m and T by n we get the theorem statement. ■

We get a large potential increase in the number of arrays thanks to parallel programming, but still zero asymptotic capacity.

B. Parallel programming in the rows and columns

We saw in the previous subsection that the parallel programming in the rows manages to overcome all cells in conflict in the rows, but not in the columns. Thus, we add more flexibility in the programming order and now assume that it is possible to program in parallel the memristors both in the rows and the columns, in any order we wish. Thus when programming memristors in the same column, the write sneak paths can only affect the cells outside the programmed column and the same principle holds for the rows.

Therefore, if memristors in conflict are in the same row or in the same column, then it is possible to resolve this conflict by simply programming them in parallel. However, the patterns which we still cannot resolve are the ones in which a memristor is in conflict with another memristor on its row and another one on its column. This will be proved in the next theorem.

Theorem 11. *Assume an array A has a memristor which is in conflict with two more memristors, one in the same row and one in the same column. Then, it is not possible to successfully program its cells.*

Proof: Assume the memristor in position (i, j) is in conflict with two more memristors, one in position (i, j') and another one in position (i', j). It is not possible to program all three cells in parallel. Thus, in any programming order at least one of the three cells will be last and cause an error. ■

Lastly, this condition is sufficient (proof omitted).

Theorem 12. *Assume an array A has no memristor which is in conflict with two more memristors, one in the same row and one in the same column. Then, it is possible to successfully program its cells.*

VI. Conclusion

We characterized the constraints memristor arrays need to satisfy for different programming models. Important future work is to tighten the counting of arrays satisfying these constraints, and derive similar counts for the last model of parallel row and column programming.

VII. Acknowledgement

This work was supported in part by the Intel ICRI-CI center, by the Israel Science Foundation (ISF), and the German-Israeli Foundation (GIF).

References

[1] Y. Cassuto, S. Kvatinsky, and E. Yaakobi, “Information-theoretic sneak path mitigation in memristor crossbar arrays,” submitted to *IEEE Transactions on Information Theory*, 2013.

[2] Y. Cassuto, S. Kvatinsky, and E. Yaakobi, “On the channel induced by sneak-path errors in memristor arrays,” *Proc. of the International Conference on Signal Processing and Communication*, pp. 1–6, July 2014.

[3] Y. Cassuto, S. Kvatinsky, and E. Yaakobi, “Sneak-path constraints in memristor crossbar arrays,” in *Proc. IEEE International Symposium on Information Theory*, pp. 156–160, Istanbul, Turkey, July 2013.

[4] L.O. Chua, “Memristor the missing circuit element,” *IEEE Transactions on Circuit Theory*, vol. 18, no. 5, pp. 507–519, Sep. 1971.

[5] L.O. Chua, “Resistance switching memories are memristors,” *Applied Physics A: Materials Science & Processing*, vol. 102, no. 4, pp. 765–783, March 2011.

[6] X. Cong, D. Xiangyu, N. P. Jouppi, and Y. Xie, “Design implications of memristor-based RRAM cross-point structures,” *Proceeding of Design, Automation & Test in Europe Conference & Exhibition*, pp. 1–6, March 2011.

[7] S. Kvatinsky, E.G. Friedman, A. Kolodny, and U.C. Weiser, “The desired memristor for circuit designers,” *IEEE Circuits and Systems Magazine*, vol. 13, no. 2, pp. 17–22, 2013.

[8] S. Kvatinsky, N. Wald, G. Satat, E. G. Friedman, A. Kolodny, and U. C. Weiser, “Memristor-based material implication (IMPLY) logic: design principles and methodologies,” *IEEE Transactions on Very Large Scale Integration (VLSI)*, vol. 22, no. 10, pp. 2054–2066, October 2014.

[9] H. Li et al., “Write disturb analyses on half-selected cells of cross-point RRAM arrays,” *Proc. of the IEEE International Reliability Physics Symposium*, pp. MY.3.1-MY.3.4, June 2014.

[10] P.P. Sotiriadis, “Information capacity of nanowire crossbar switching networks,” *IEEE Transactions on Information Theory*, vol. 52, no. 7, pp. 3019–3032, July 2006.

[11] D. Strukov, G. Snider, D. Stewart, and S. Williams, “The missing memristor found,” *Nature*, vol.,453, pp. 80–83, May 2008.

[12] P.O. Vontobel, W. Robinett, P.J. Kuekes, D.R. Stewart, J. Straznicky, and S. Williams, “Writing to and reading from a nano-scale crossbar memory based on memristors,” *Nanotechnology*, vol. 20, October 2009.