Permanent and Transient Fault Tolerance for Reconfigurable Nano-Crossbar Arrays

Onur Tunali and Mustafa Altun

***Abstract*—This paper studies fault tolerance in switching reconfigurable nano-crossbar arrays. Both permanent and transient faults are taken into account by independently assigning stuck-open and stuck-closed fault probabilities into crosspoints. In the presence of permanent faults, a fast and accurate heuristic algorithm is proposed that uses the techniques of index sorting, backtracking, and row matching. The algorithm's effectiveness is demonstrated on standard benchmark circuits in terms of runtime, success rate, and accuracy. In the presence of transient faults, tolerance analysis is performed by formally and recursively determining tolerable fault positions. In this way, we are able to specify fault tolerance performances of nano-crossbars without relying on randomly generated faults that is relatively costly regarding that the number of fault distributions in a crossbar grows exponentially with the crossbar size.**

***Index Terms*—Nano-crossbars; Fault Tolerance; Switching Arrays; Permanent and Transient Faults/Defects.**

I. Introduction

NANO-CROSSBAR arrays have emerged as a strong candidate technology to replace CMOS in near future [2] [3]. They are regular and dense structures, and fabricated by exploiting self-assembly as opposed to purely using lithography based conventional and relatively costly CMOS fabrication techniques [4] [5]. Currently, nano-crossbar arrays are fabricated such that each crosspoint can be used as a conventional electronic component such as a diode, a FET, or a switch [6] [7]. This is a unique opportunity that allows us to integrate well developed conventional circuit design techniques into nano-crossbar arrays. However, as expected, the integration comes with some challenges and fault/defect tolerance is one of the significant ones. Fault rates are much higher for nano-crossbars compared to those of conventional CMOS circuits [8] [9]. Therefore developing efficient fault tolerance techniques for nano-crossbars is a must and the main motivation of this study. In this study, we examine reconfigurable crossbar arrays by considering randomly occurred stuck-open and stuck-closed crosspoint faults. This is illustrated in Figure 1. Our fault tolerance approach is based on an assumption that a crossbar input can be used for multiple crossbar outputs (broadcasting allowed) that fits Boolean logic applications. On the other hand, especially for memory applications a crossbar input is strictly used for only one output that necessitates different fault tolerance approaches [10] [11].

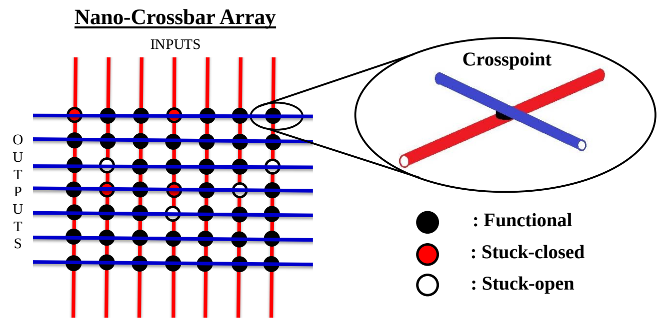

Fig. 1. Nano-crossbar array with faulty/defective crosspoints.

TABLE I
PERMANENT VERSUS TRANSIENT FAULTS.

Permanent Faults	Transient Faults
• Occurring mostly in fabrication • Tolerated in design phase • Tolerated by reconfigurability (mapping) and redundancy	• Occurring in field • Tolerated in use phase • Tolerated by redundancy

We propose distinct approaches for permanent and transient faults regarding their exclusive natures as shown in Table I. In the presence of permanent faults, tolerance is achieved by mapping target Boolean functions on a defective crossbar using crossbar row and column permutations. This is an NP-complete problem [12]. For the worst-case scenario, implementing a target function with an $N \times M$ crossbar requires $N!M!$ permutations; computing time quickly grows to intractable levels with the crossbar size. To tackle this problem, several approaches have been proposed in the literature that can be classified into two main categories: defect-unaware and defect-aware approaches.

Defect-unaware algorithms aim to find the largest possible $k \times k$ defect-free sub-crossbar from a defective $N \times N$ crossbar where $k \leq N$ [13] [14] [15]. Detailed yield analysis of these algorithms shows a common shortcoming: the algorithms are inefficient for high fault rates – obtained k values are much smaller than N [15]. When $N = 250$ and the fault rate is 15% that is a reasonable value for nano-arrays, the fastest algorithms find k values as high as 30 [15]. It means that only 1% of the crossbar can be used. In this regard, defect-aware algorithms perform much more satisfactorily [16] [17] [18]. A valid mapping is generally found using a 1.5 times larger row and column sizes than the optimal sizes. Note that for a specific target function, the larger the crossbar, the

* This work is supported by the EU-H2020-RISE project NANOxCOMP #691178 and the TUBITAK-Career project #113E760.

* O. Tunali and M. Altun are respectively with the Department of Nanoscience and Nanoengineering and the Department of Electronics and Communication Engineering, Istanbul Technical University, Istanbul, Turkey, 34469.

* E-mails: {onur.tunali, altunmus}@itu.edu.tr

* A preliminary version of this paper appeared in [1]

easier to find a valid mapping due to an increase in solution space. Therefore it is challenging, as well as desired for area considerations, to find a mapping with optimal size crossbars. We satisfy this with our heuristic defect-aware algorithm.

Defect-aware algorithms which use graph based heuristics, transform the mapping problem into a graph isomorphism problem [19] [20] [16]. An initial input assignment is made to prune the permutation space. However, in case of an unfavourable assignment, the number of reconfigurations needed to find a valid mapping increases drastically . Additionally, the runtime quickly grows beyond practical limits, especially for large-scale target functions. Other algorithms based on integer linear programming also suffer from runtime inefficiency for large-scale functions [21] [18]. Apart from the mentioned methods, a considerably fast memetic algorithm is proposed to tackle this problem. [22]. Here the drawback is that the starting conditions affect the results significantly. As an example, experimental results presented in [22] show as large as a 25 times difference in runtimes for the same size target functions. Our proposed algorithm works considerably faster compared to the algorithms in the literature with nearly steady runtime values for the same size target functions. To our knowledge no other algorithm is able to find a valid mapping for large benchmarks such as “table5” and “t481” with up to 15% fault rates. Additionally, the proposed algorithm shows 99% accuracy in accordance with the results of an exhaustive search algorithm.

Our algorithm performs sorting to avoid disadvantageous initial appointments and reduce unnecessary reconfigurations. For this purpose, matrix and index representations of target functions and defective crossbars are obtained. Sorted matrices are matched using one dimensional array matchings that makes the mapping problem to be solved with mere multiplication operations. Backtracking is also performed to improve accuracy.

Although permanent fault tolerance of nano-crossbar arrays have been thoroughly studied in the literature, transient faults are not adequately emphasized. Redundancy based approaches are proposed to tolerate transient faults by exploiting techniques including majority voting, hardening, and fault masking [17] [23] [24] [25] [26] . For these studies, the main goal is to find an efficient method of adding extra redundancies to correct/detect single or multiple faults while optimizing the area overhead. In this study, we do not aim to correct faults; instead we aim to determine tolerable fault positions in advance without increasing area. We adopt a formal approach instead of randomly generating faults and checking whether the faults ruin the crossbar functionality. We determine equivalent logic functions of a target function that denotes the positions of tolerable faulty switches. We show that iff faults occur on these positions, the crossbar still implements the correct function. In other words, we show that it is possible to tolerate transient faults without adding extra redundancies. In this way, we are able to specify fault tolerance performance without relying on a Monte Carlo simulation that is relatively costly regarding that the number of fault distributions in a crossbar grows exponentially with the crossbar size.

Our method can be used for the above mentioned studies to manipulate redundancies using the obtained tolerable fault positions. Additionally, the obtained equivalent Boolean functions can be used generally for logic equivalence problems.

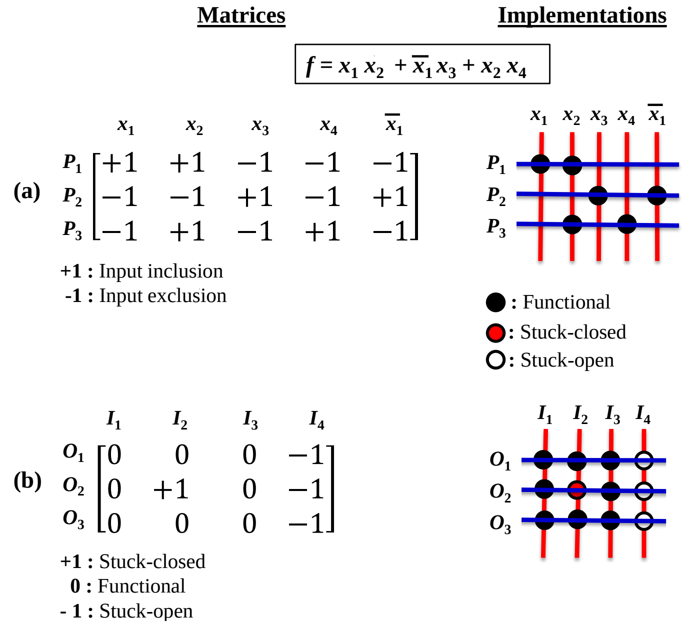

Fig. 2. Matrix representations and crossbar implementations for (a) a function f and (b) a defective crossbar.

Organization of the paper is as follows. In Section II, we present the proposed fault tolerance algorithm for permanent faults. In Section III, we explain transient faults, their reliability analysis, and eventually a performance calculation method. In Section IV, we present experimental results and elaborate on them. In Section V, we discuss our contributions and future works.

A. Definitions

In this section, we explain key concepts used throughout the paper for both permanent and transient faults.

Definition 1: Consider k independent **Boolean variables**, $x_1, x_2, \ldots, x_k$. **Boolean literals** are Boolean variables and their complements, i.e., $x_1, \bar{x}_1, x_2, \bar{x}_2, \ldots, x_k, \bar{x}_k$.

Definition 2: A **product (P)** is an AND of literals, e.g., $P = x_1\bar{x}_3x_4$. A **sum-of-products (SOP)** expression is an OR of products.

Definition 3: A **prime implicant (PI)** of a Boolean function f is a product that implies f such that removing any literal from the product results in a new product that does not imply f.

Definition 4: An **irredundant sum-of-products (ISOP)** expression is an SOP expression, where each product is a PI and no PI can be deleted without changing the Boolean function f represented by the expression.

Definition 5: A **sum (S)** is an OR of literals, e.g., $S = x_1 + \bar{x}_3 + x_4$. A **product-of-sums (POS)** expression is an AND of sums.

Definition 6: **Function matrix (FM)** is a representation of a Boolean function in SOP form such that the function’s literals and products are appointed to the matrix columns and rows, respectively. If a literal occurs in a product, it is denoted with +1; otherwise -1 is assigned. Figure 2 (a) shows an example of a function matrix.

Definition 7: **Crossbar matrix (CM)** is a representation of a crossbar array such that functional switches of crossbars are denoted with 0; defective stuck-closed and stuck-open switches are denoted with +1 and -1, respectively. Figure 2 (b) shows an example of a crossbar matrix by considering stuck-closed and stuck-open faults.

Definition 8: **Logic inclusion ratio (IR)** is defined as a ratio of the number of +1's, corresponding to used switches, to the total number of elements, +1's and -1's, in a function matrix. As an example, consider the function matrix in Figure 2 (a). Here, the number of +1's or the number of used switches is 6, so **IR** = 6/15.

II. Permanent Fault Tolerance

We aim to find out a valid mapping, namely a correct assignment of literals and products of a target function to inputs and outputs of a given crossbar having permanent faults. Positions of the faults are known, represented by a crossbar matrix, prior to mapping. We consider randomly distributed stuck-closed and stuck-open faults at crosspoint switches; wire breakdowns and bridging faults are not considered in this study.

In case of having a defect-free crossbar, every assignment produces a valid mapping. Figure 3 (a) shows two different assignments resulting in valid mappings for a target function f. However, finding a valid mapping for a defective crossbar requires trials of different assignments. This is illustrated in Figure 3 (b). While the assignment in the upper part produces an incorrect mapping since x_1 of P_1 is positioned on a stuck-open fault, the assignment in the lower part is correct resulting in a valid mapping. The main purpose of our algorithm is to find a correct assignment or a valid mapping ; a formal problem definition is given as follows.

Problem Definition: Consider different assignments of literals (x's) to inputs and products (P's) to outputs. An input array I[$x_i, \cdots, x_j$] and an output array O[$P_i, \cdots, P_j$] are defined such that i^{th} elements of the arrays are the assigned literal and product to the i^{th} crossbar input and output, respectively. The proposed algorithm yields input and output arrays that establish a valid mapping or a correct assignment. As an example, the correct assignment in the lower part of Figure 3 (a) has I = [x_1 x_3 x_2 x_4] and O = [P_2 P_1 P_3].

Our algorithm fundamentally uses index representations of function and crossbar matrices as well as row/column permutations and matchings. These concepts are explained as follows.

A. Preliminaries

Row index: The number of +1, 0, or -1 valued elements in a matrix row. For example, the row represented by P_1 in Figure 4 has a row index of 3 for a chosen value of +1.

Column index: The number +1, 0, or -1 valued elements in a matrix column. For example, the column represented by x_1 in Figure 4 has a column index of 1 for a chosen value of -1.

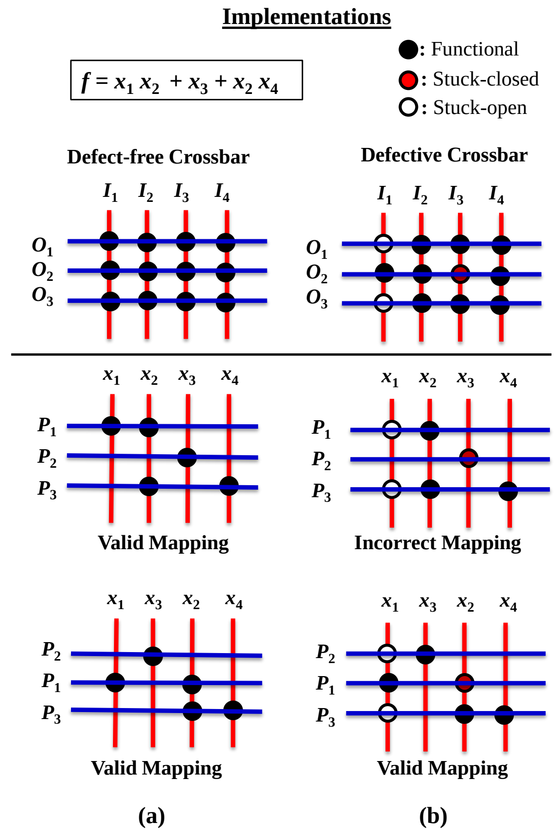

Fig. 3. Logic function implementations for (a) a defect-free crossbars (b) a defective crossbars with assignments.

Row index set: A set of all row indices of a matrix for a chosen value of +1, 0, or -1. In Figure 4, rows represented by P_1, P_2, and P_3 have row indices of 1, 2, and 2, respectively, for a chosen value of -1. So its row index set is $I_{R,F} = \{1, 2, 2\}$ where R stands for *row* and F stands for *function*.

Column index set: A set of all column indices of a matrix for a chosen value of +1, 0, or -1. In Figure 4, columns represented by x_1, x_2, x_3, and x_4 have column indices of 2, 2, 1, and 2, respectively, for a chosen value of +1. So its column index set is $I_{C,F} = \{2, 2, 1, 2\}$ where C stands for *column* and F stands for *function*.

Row/Column permutation: In order to find a valid mapping, defective switches of a crossbar matrix which are denoted as +1's (stuck-closed) and -1's (stuck-open) must be matched with +1's (used) and -1's (unused), respectively in a function matrix. Here, an important property is that row and column permutations in the function matrix do not alter the implemented function. This is an important reconfigurability feature for fault tolerance as illustrated in Figure 4.

Row matching with Hadamard product: In order to match two rows from function and crossbar matrices, we use Hadamard product by performing element-by-element multiplication that is similar to an inner product operation used for vectors. If there is any negative valued element in the resulting matrix then there is no matching; otherwise there is a valid matching. Note that functional switches (denoted with 0) in the crossbar matrix can be always matched with either +1's

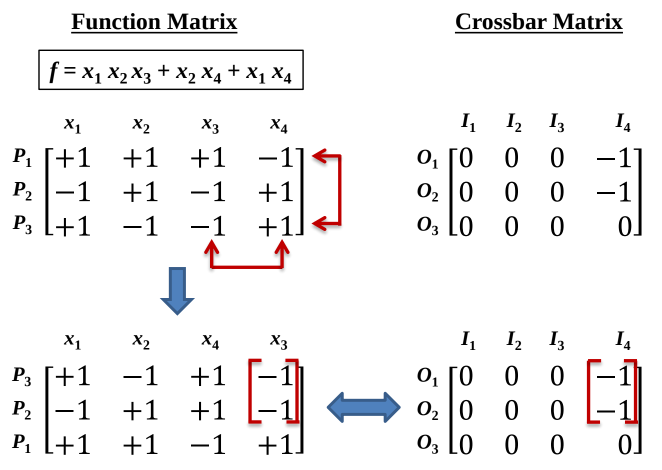

Fig. 4. Row and column permutations of the function matrix to obtain a valid mapping in case of having stuck-open faults.

TABLE II
ELEMENT COMPATIBILITY OF FUNCTION MATRIX (FM) AND CROSSBAR MATRIX (CM).

FM_{ik}	CM_{ik}	$FM_{ik} \times CM_{ik}$	Matching
+1	+1	+1	✓
+1	0	0	✓
-1	0	0	✓
-1	-1	+1	✓
+1	-1	-1	×
-1	+1	-1	×

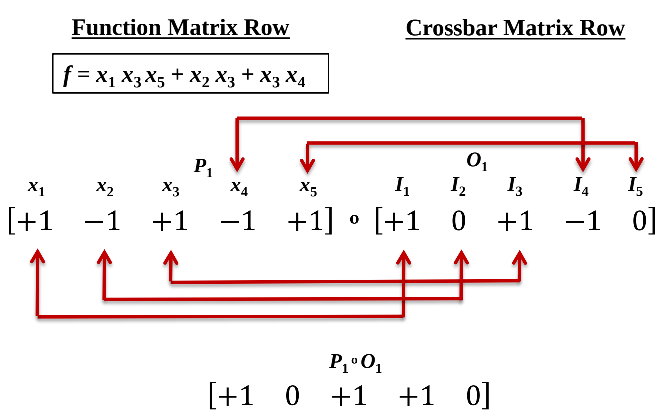

Fig. 5. Hadamard product of row matrices represented by P_1 and O_1. The resulting matrix has no negative element; there is a valid matching.

or -1's in the function matrix. However, +1's and -1's in the crossbar matrix can only be matched with +1's and -1's in the function matrix, respectively. This is illustrated in Table II. Additionally, Figure 5 shows an example for a valid matching between the first rows of the matrices in case of having stuck-closed and stuck-open faults.

B. Proposed Algorithm

The outline of our four-step algorithm is shown in Figure 6. Step 1 starts with obtaining index sets of function and crossbar matrices. Using the sets, crossbar matrices are sorted according to either stuck-closed (+1) or stuck-open (-1) faults such that

Input: Function and crossbar (defective) matrices of size $N \times M$

Output: "YES" with valid input and output assignments if the matrices are matched; "NO" otherwise

Step 1 **Sorting**: Sort function and crossbar matrices using row and column index sets according to either stuck-closed (+1) or stuck-open (-1) faults.

Step 2 **Matching**: Starting from the top row in the function matrix, perform matching with Hadamard product by advancing search from the top row to the bottom row of the crossbar matrix. If all of the function rows are matched then return "YES".

Step 3 **Backtracking**: If no matching is found for a function row then search previously matched crossbar rows from top to bottom. If a matching is found then repeat Step 2 by excluding the already matched rows.

Step 4 **Repeating**: If no matching is found then repeat Step 2 (and Step 3) for *PL*=3000 times by randomly applying a pairwise crossbar column permutation. If a matching cannot be found under *PL* trials then return "NO".

Fig. 6. Outline of the proposed algorithm.

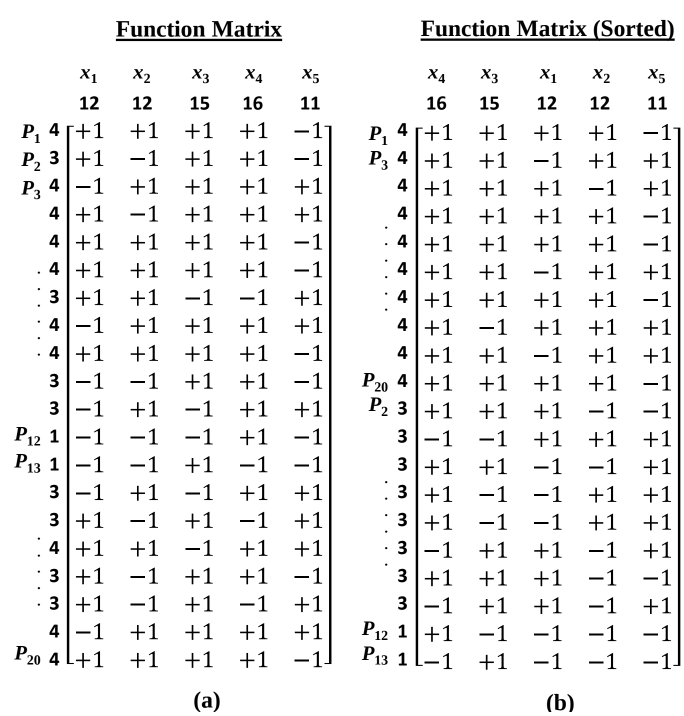

Fig. 7. According to stuck-closed faults(+1) (a) function matrix and (b) its sorted form.

rows and columns with the most defective elements are aligned to the top and the left sides, respectively. Function matrices are sorted in the same manner as shown in Figure 7. Using sorted matrices significantly reduce the matching workload in the next step. Note that although we treat stuck-closed and stuck-open faults separately throughout this study, our algorithm works properly in case having both fault types in crossbars.

$$
\begin{array}{c}
\textbf{Function Matrix (Sorted)} \\
\begin{array}{c}
R_1 \\ R_2 \\ \\ \cdot \\ \cdot \\ \\ \\ \\ \\ \cdot \\ \cdot \\ \\ R_{13} \\ R_{14} \\ \\ \\ \\ \\ \\ \\
\end{array}
\begin{bmatrix}
+1 & +1 & +1 & +1 & -1 \\
+1 & +1 & -1 & +1 & +1 \\
+1 & +1 & +1 & -1 & +1 \\
+1 & +1 & +1 & +1 & -1 \\
+1 & +1 & +1 & +1 & -1 \\
+1 & +1 & -1 & +1 & +1 \\
+1 & +1 & +1 & +1 & -1 \\
+1 & -1 & +1 & +1 & +1 \\
+1 & +1 & -1 & +1 & +1 \\
+1 & +1 & +1 & +1 & -1 \\
+1 & +1 & +1 & -1 & -1 \\
-1 & -1 & +1 & +1 & +1 \\
+1 & +1 & -1 & -1 & +1 \\
+1 & -1 & -1 & +1 & +1 \\
+1 & -1 & -1 & +1 & +1 \\
-1 & +1 & +1 & -1 & +1 \\
+1 & +1 & +1 & -1 & -1 \\
-1 & +1 & +1 & -1 & +1 \\
+1 & -1 & -1 & -1 & -1 \\
-1 & +1 & -1 & -1 & -1
\end{bmatrix}
\end{array}
\qquad
\begin{array}{c}
\textbf{Crossbar Matrix (Sorted)} \\
\begin{bmatrix}
+1 & +1 & 0 & +1 & 0 \\
+1 & +1 & +1 & 0 & +1 \\
0 & +1 & 0 & 0 & +1 \\
+1 & 0 & 0 & +1 & +1 \\
+1 & 0 & 0 & +1 & +1 \\
+1 & +1 & 0 & 0 & 0 \\
0 & 0 & +1 & 0 & +1 \\
0 & +1 & 0 & 0 & +1 \\
+1 & 0 & +1 & 0 & 0 \\
0 & +1 & +1 & 0 & 0 \\
+1 & 0 & 0 & 0 & +1 \\
0 & +1 & 0 & 0 & +1 \\
+1 & 0 & +1 & 0 & 0 \\
0 & 0 & 0 & +1 & +1 \\
+1 & 0 & 0 & 0 & 0 \\
0 & 0 & +1 & 0 & 0 \\
0 & 0 & +1 & 0 & 0 \\
0 & +1 & 0 & 0 & 0 \\
0 & 0 & +1 & 0 & 0 \\
0 & +1 & 0 & 0 & 0
\end{bmatrix}
\begin{array}{c}
- \\ 3 \\ - \\ 2 \\ 6 \\ 1 \\ 8 \\ 9 \\ 4 \\ 5 \\ 13 \\ - \\ 7 \\ 12 \\ 10 \\ 11 \\ - \\ - \\ - \\ -
\end{array}
\end{array}
$$

Fig. 8. An example of backtracking for the row R_{14}.

Step 2 performs row by row matching between the sorted matrices advancing from top to bottom. For the matched matrices, the number of columns is always less than or equal to the number of rows. In case, a function or a crossbar matrix does not satisfy this, it is transposed. The reason of this operation is to decrease the number of trials in Step 4.

If a function matrix row can not be matched with any of the unmatched crossbar matrix rows then the algorithm proceeds to Step 3. Figure 8 illustrates an example; numbers in red assigned to the crossbar matrix rows represent the orders of the corresponding matched rows in the function matrix. Every row of the function matrix until the 14^{th} row R_{14} is matched with a row in the crossbar matrix. Since R_{14} cannot be matched with any of the unmatched rows, backtracking starts by checking the previously matched crossbar rows from top to bottom. This results in a matching with the 4^{th} row followed by performing Step 2 by excluding the matched rows. Note that after backtracking R_2 becomes unmatched and is to be matched with the unmatched crossbar matrix rows. This prevents a recursive character that would cause a significant computational load.

In case backtracking does not result in a valid matching, the algorithm proceeds to Step 4 with repeating Step 2 (and Step 3) at most PL (Permutation Limit) times. Here, column permutations are randomly applied. Note that Step 4 is used as a contingency plan to maintain certain performance metrics including accuracy and success rate (Psucc). Accordingly, the value of PL is determined. In this study, we aim to maintain minimum of 95% success rate. For this purpose, we randomly generate function and crossbar matrices for different crossbar sizes with a fault rate of 15% that is an accepted upper limit for nano-crossbars [27] and an inclusion ratio of 40% that is a typical average value for benchmark functions. The results using optimal size crossbars and 1.5 larger sizes than the optimal ones are given in Figure 9 (a) and Figure 9 (b), respectively. Both graphs clearly show a steep increase after PL exceeds 2000. It means that selecting PL considerably larger than 2000 does slightly improve the success rate of the algorithm while it would increase the runtime significantly. We select PL = 3000 in this study. Indeed, our algorithm proceeds to Step-4 only for very small portion of benchmark simulations that are thoroughly explained in Section IV.

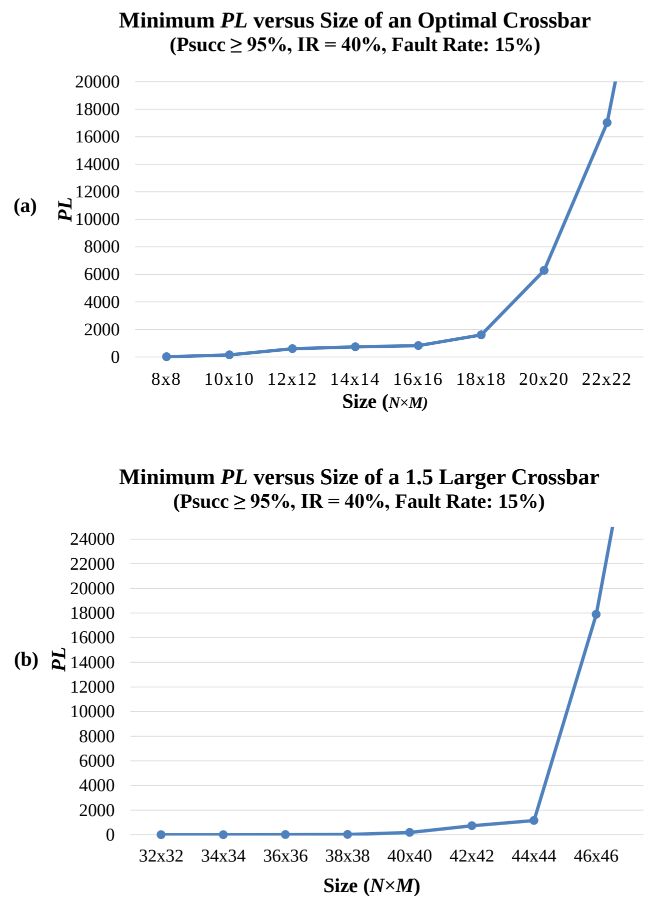

Fig. 9. Minimum permutation limit PL needed to achieve 95% success rate versus size $N \times M$ for (a) optimal size crossbars (b) 1.5 larger size crossbars.

Since permutations are performed column wise, we expect much stronger relation of PL with the number of columns M compared to the number of rows N. The relation between PL and M can be relatively examined with the following probability analysis. Consider function and crossbar matrix rows to be matched. In case of having stuck-closed faults with a fault probability of p_f, probability of having a valid matching between these rows can be found as:

$$Pr_m(M, a, b) = \frac{\binom{M-a}{f_1-a}}{\binom{M}{b}}$$

where $a = p_f \cdot M$ and $b = IR \cdot M$ represent expected values for the number of 1's in crossbar and function rows, respectively. Additionally, probability of having a valid matching after performing a pairwise permutation (initially no matching) can be found as

$$Pr_p(M, a, b) = \frac{a \cdot [b-a+1] \cdot \binom{M-a}{b-a+1}}{\binom{M}{2} \cdot [\binom{M}{b} - \binom{M-a}{b-a}]}.$$

By considering constant IR and p_f values, we can comment that 1) increasing M makes Pr_p decrease; 2) decreasing Pr_p

reduces the effectiveness of performing a permutation 3) PL is negatively correlated with Pr_p; 4) if Pr_p decreases to relatively small levels then increasing PL would not significantly contribute in finding a valid matching that is also verified by the results in Figure 9.

A pseudo code of the proposed heuristic algorithm is depicted in Algorithm below. The algorithm yields input and output arrays that establish a valid mapping or a correct assignment of a target function into a defective crossbar.

C. Performance Evaluation

Our algorithm uses a constant permutation for one dimension (column) and advancing through the other one (row) that reduces the number of operations for finding a valid mapping [20] [23]. Instead of using conventional two dimensional matchings of matrices, our algorithm performs considerably faster one dimensional matrix row matchings. Our motivation is that the main problem of mapping target functions has many different solutions. Therefore probable information lost in one dimensional check can be easily compensated; backtracking and repeating is also for this purpose. Here, an important factor is the relation between logic inclusion ratio (IR) and fault rate. For a constant IR around 40%, a typical average value for standard benchmark functions, an increase in the fault rate especially beyond 25% significantly reduces the number of mapping solutions that worsens the performance of our algorithm. For fault rates below 25%, our algorithm works satisfactorily in terms of both runtime and accuracy with surpassing related algorithms in the literature. Our algorithm's performance is also justified with a complexity analysis as follows and detailed experimental results in Section IV.

Consider a function/crossbar matrix with a size of $N \times M$ where $N \geq M$. The number of initial operations for every row checking is M for multiplication plus M for comparison, so in total of $2M$. Additionally, each function row is matched with N crossbar rows, so $2M \cdot N$ operations are needed. In case of backtracking, another N rows need to be checked that results in $2M \cdot [N + N]$ operations. For all of the function rows, there are $N \cdot [2M \cdot [N + N]]$ operations. Considering PL trials in the last step of the algorithm, the number of operations become $(PL + 1) \cdot [2 \cdot M \cdot [N + N]]$. If we select a constant number for $PL = 3000$ that is independent of M, our algorithm works in $O(M \cdot N^2)$ time. Of course, for the worst-case scenario where $M!$ permutations are performed, the complexity becomes factorial.

III. Transient Fault Tolerance

Regarding the probabilistic and the continuous feature of transient faults in time domain, their tolerance can not be achieved by applying the same technique used for permanent faults that is based on fault identification followed by reconfiguration. Transient fault tolerance is purely based on redundancy. For nano-crossbar arrays, redundancy is correlated with the logic inclusion ratio (IR) as well as the used sum-of-product representations of target functions.

Similar to permanent faults, we consider stuck-open and stuck-closed transient faults that are treated separately. We

Algorithm Heuristic Algorithm

```
Input: Function Matrix (FM), Crossbar Matrix (CM), and Permutation Limit PL
Output: I[i] and O[i] arrays

function INDEX_SORT(M)
    I_{R,M} ← Row Index Set according to the selected fault type
    I_{C,M} ← Column Index Set according to the selected fault type
    Sort I_{R,M} descending
    Sort I_{C,M} descending
    row_permutation ← I_{R,M}
    column_permutation ← I_{C,M}
    M ← M[row_permutation, column_permutation]
    return M
end function

INDEX_SORT(FM)
I[i] ← column_permutation of FM
INDEX_SORT(CM)
for t=1 to PL do
    O[i] = []
    if t > 1 then
        change column_permutation
        I[i] ← column_permutation
    end if
    for k=1 to N do
        F_k ← kth row of FM
        for j=1 to N and O[j] = [] do
            C_j ← jth row of CM
            if F_k .* C_j ≥ 0 then
                O[k] = j
                break
            end if
        end for
        if no matching then                ▷ Backtracking process
            for j in O[i] do
                C_j ← jth row of CM
                if F_k .* C_j ≥ 0 then
                    O[k] = j
                    break
                end if
            end for
        end if
        if matching found then             ▷ O[i] changed
            F_m ← previously matched row of FM
            for j=1 to N and O[j] = [] do
                C_j ← jth row of CM
                if F_m .* C_j ≥ 0 then
                    O[m] = j               ▷ Rematching process
                    break
                end if
            end for
        end if
        if no matching found for F_m then
            break                          ▷ column_permutation changes
        end if
    end for
end for
```

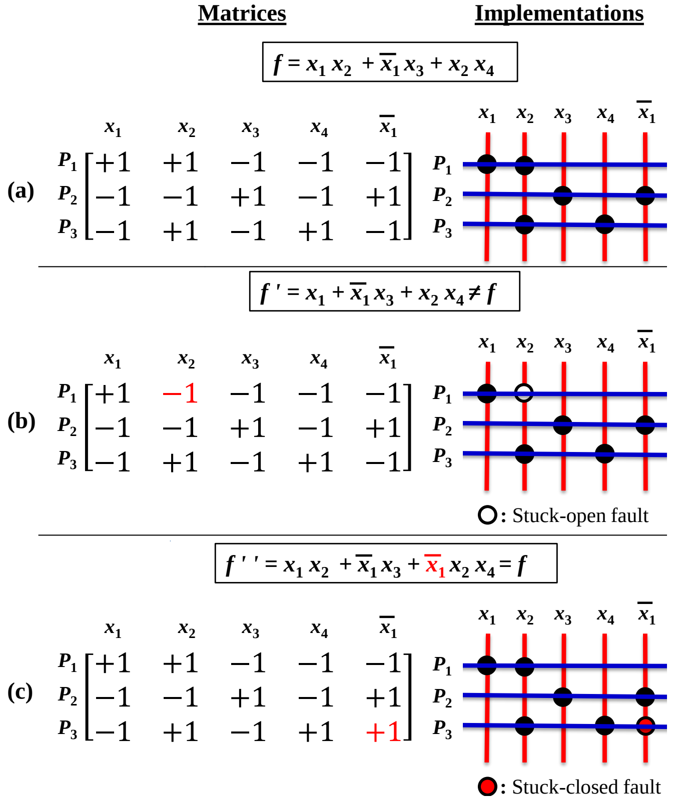

Fig. 10. Implementations in the presence of (a) no faults (b) stuck-open faults, and (c) stuck-closed faults.

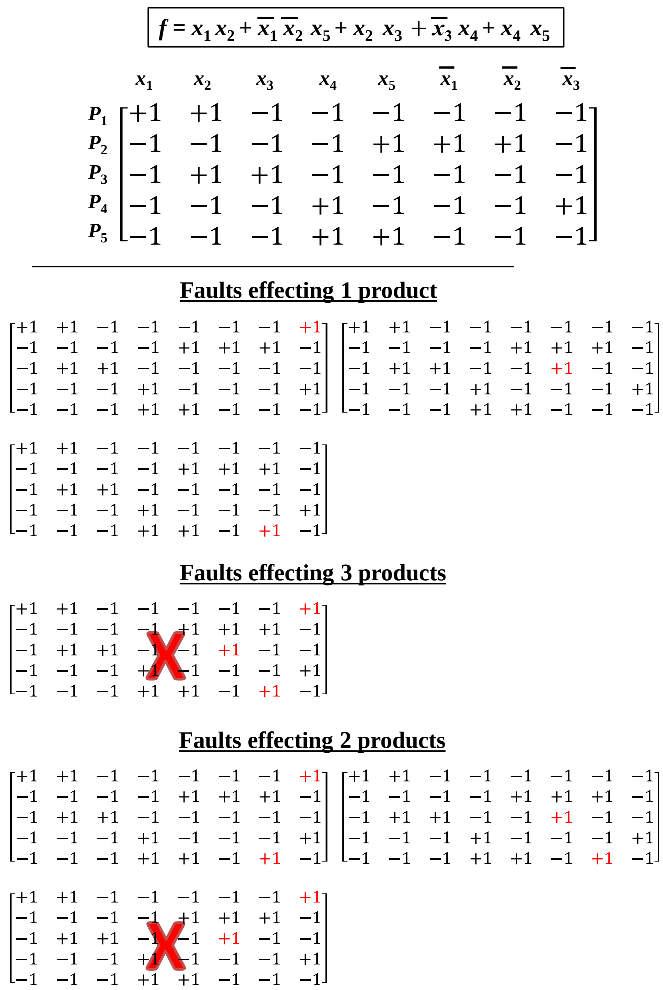

Fig. 11. Tolerable and intolerable (with red crosses) fault positions.

suppose that target functions are implemented in irredundant sum-of-products (ISOP) forms to minimize the number of used switches for cost optimization in fabrication. We analyse fault tolerance performance of nano-crossbar arrays by considering the specifics of target functions. Figure 10 shows an example. A given target function f in ISOP form is implemented with a fault-free crossbar shown in Figure 10 (a). When a stuck-open fault occurs on a used switch (denoted with +1's) as shown in Figure 10 (b), the corresponding literal is erased from the target function and the corresponding matrix element becomes -1. In this example, since the new function f' is not equal to the original function f, the fault cannot be tolerated. When a stuck-closed fault occurs on an unused switch (denoted with -1's) as shown in Figure 10 (c), the corresponding literal is added to the target function and the corresponding matrix element becomes +1. Here, the new function f'' is equal to f, so the fault is tolerated.

A. Stuck-Open Faults

Stuck-open faults are tolerated iff they occur on unused switches. Faults on used switches change the implemented functions. Since we use ISOP forms of target functions consisting of prime implicants, by definition removing any literal from a prime implicant results in a new function. Fault tolerance performance FT_{so} of an $N \times M$ crossbar can be directly calculated by using

$$FT_{so} = (1 - p_{so})^{N \cdot M \cdot IR}$$

where p_{so} is an independent stuck-open fault probability of each switch and IR is the logic inclusion ratio. Note that our analysis for stuck-open faults is applicable for both single-output and multi-output functions.

B. Stuck-Closed Faults

We show that along with all stuck-closed faults occurring on used switches, faults on unused switches can also be tolerated. This is illustrated in Figure 11 with a brief summary of our tolerance analysis method. We determine all possible positions of tolerable faults on unused switches in the crossbar. These positions, represented by added +1's in red in Figure 11, are determined recursively. First, tolerable fault positions in single rows are determined. For the example in Figure 11, among 5 rows representing 5 products of the target function, 3 of them have the positions. Therefore there are 3 matrices showing tolerable fault positions. Analysing the first matrix at the upper-left corner, we conclude that a stuck-closed fault in the first row at the right end of the crossbar can be tolerated; $f' = x_1 x_2 \overline{x_3} + \overline{x_1}\ \overline{x_2} x_5 + x_2 x_3 + \overline{x_3} x_4 + x_4 x_5 = f$. The same is valid for the second and the third matrices as well. Next, we determine tolerable fault positions simultaneously occurring in all of the three rows. For the example in Figure

11, there is no solution for this case, so we proceed to next steps by decreasing the number of rows that the faults are seen until there is a solution. Among $\binom{3}{2} = 3$ probable row pairs with tolerable fault positions, 2 of them have solutions.

In order to find all possible positions of tolerable faults, we exploit logic equivalences of Boolean expressions. Consider a given target function $f = P_1 + ... + P_m$ in ISOP form. Stuck-closed faults on unused switches add literals to the corresponding products that results in a new function named f_t. Our main purpose is finding all f_t's such that $f_t = f$. Two examples of f_t's corresponding to the top two matrices in Figure 11 are $f_{t_1} = x_1x_2\overline{x_3}+\overline{x_1}\ \overline{x_2}x_5+x_2x_3+\overline{x_3}x_4+x_4x_5$ and $f_{t_2} = x_1x_2 + \overline{x_1}\ \overline{x_2}x_5 + \overline{x_1}x_2x_3 + \overline{x_3}x_4 + x_4x_5$. Added products of literals, shown in red, are named as P_{t_i}'s where i represents the corresponding product number. As an example, f_{t_1} has $P_{t_1} = \overline{x_3}$; f_{t_2} has $P_{t_3} = \overline{x_1}$. A general form of f_t can be represented as

$$f_{t_{\{i,..,k\}}} = P_1 + ... + P_iP_{t_i} + ... + P_kP_{t_k} + ... + P_m$$

where the subscript of f, $\{i,..,k\}$ set shows which products have added literals.

Our method for finding all $f_{t_{\{i,..,k\}}} = f$'s has two steps. In the first step, we determine tolerable fault positions affecting single products. We obtain all $f_{t_{\{i\}}}$'s and corresponding P_{t_i}'s, $1 \le i \le m$ for which a necessary and sufficient condition is given in Theorem 1. In the second step, we first construct an f_t such that it has all P_{t_i}'s obtained in the first step. If the f_t is equal to the target function f then we are done with finding all tolerable fault positions; no further steps are necessary as justified by Theorem 2. If the functions are not equal to each other then we advance through decrementing the number of products affected by faults. We repeat this until the equivalence(s) are satisfied.

As a core property used in the theorems, we first present the following lemma.

Lemma 1: Consider $f_1 = P_1+...+P_i+...+P_m, 1 \le i \le m$, in SOP form and $f_2 = S_1 \cdots S_k$ in POS form. Additionally, f_3 in SOP form is obtained by removing a sum S_j, $1 \le j \le k$, from f_2. If $P_1 + ... + P_i \cdot f_2 + ... + P_m = f_1$ then $P_1 + ... + P_i \cdot f_3 + ... + P_m = f_1$.

Proof: It is apparent that $P_1 + ... + P_i \cdot f_3 + ... + P_m = P_1 + ... + P_i \cdot f_3 \cdot (S_j + \overline{S_j}) + ... + P_m = f_1 + P_1 + ... + P_i \cdot f_3 \cdot \overline{S_j} + ... + P_m = f_1$. ∎

Theorem 1: Consider a function $g_i = f - P_i$ in ISOP form (P_i is excluded from f). Iff P_{t_i} consists of negated forms of single-literal products in $g_i(P_i = 1)$ in ISOP form, $f = f_{t_{\{i\}}}$.

Proof: It is trivial that $f = P_i\overline{g_i}+g_i = P_i\overline{g_i(P_i = 1)}+g_i$. Here, $\overline{g_i(P_i = 1)}$ is a POS expression with sums having either single literal or multi literals. Single-literal sums are negated forms of single-literal products in $g_i(P_i = 1)$. To eliminate multi-literal sums from $P_i\overline{g_i(P_i = 1)}$, we can directly apply Lemma 1 with guaranteeing $f = f_{t_{\{i\}}}$. To prove sufficiency, we also show that each literal from P_{t_i} should correspond to a negated form of a single-literal product in $g_i(P_i = 1)$. Consider a literal l_i from P_{t_i}. From Lemma 1, we know that $f = P_il_i + g_i$. Since $f(P_i = 1) = 1$, $l_i + g_i(P_i = 1) = 1$. This necessitates having a product $\overline{l_i}$ in $g_i(P_i = 1)$ in ISOP form. ∎

Theorem 2: If $f_{t_{\{i,..,k\}}} = f$, then for $\forall x \subset \{i,..,k\}$, $f_{t_x} = f$.

Proof: The proof is a direct corollary of Lemma 1 from which we know that we can remove any literal (s) from P_{t_i}'s without disturbing the equivalence with f. ∎

Theorem 1 allows us to separately construct P_{t_i}'s showing tolerable fault positions for each P_i. Additionally, removing a literal from P_{t_i}'s does not ruin the functionality as justified by Lemma 1 that are considered in our fault tolerance analysis.

Theorem 2 significantly reduces the computing load of finding tolerable fault positions. For example, if we find for a target function f that $f_{t_{3,4,8,9}} = f$, then all tolerable fault combinations affecting products of P_3, P_4, P_8, and P_9 are known. For example, $f_{t_{3,8,9}} = f$ or $f_{t_{4,9}} = f$.

We present an example to elucidate our method.

Example 1: Consider a target function in ISOP form $f = x_1x_2x_3 + \overline{x_2}x_4x_5 + x_3x_4 + x_3\overline{x_5}$. Literal set (LS) of f is LS = $\{x_1, x_2, x_3, x_4, x_5, \overline{x_2}, \overline{x_5}\}$.

1.Step: We find faults affecting single products by exploiting Theorem 1. We only consider literals being member of LS.

$\overline{g_1(P_1 = 1)} = \overline{x_4}x_5$
$P_{t_1} = x_5$

$\overline{g_2(P_2 = 1)} = \overline{x_3}$
P_{t_2} : not a member of LS

$\overline{g_3(P_3 = 1)} = \overline{x_1}x_2x_5$
$P_{t_3} = x_2, P_{t_3} = x_5, P_{t_3} = x_2x_5$

$\overline{g_4(P_4 = 1)} = \overline{x_4}(\overline{x_1} + \overline{x_2})$
P_{t_4} : not a member of LS

2.Step: We first check whether f equals to $f_{t_{\{1,3\}}}$ having P_{t_1}, P_{t_3}. We start with P_{t_3} having the largest number of literals.
$P_{t_1} = x_5$
$P_{t_3} = x_2x_5$
$f = x_1x_2x_3 + \overline{x_2}x_4x_5 + x_3x_4 + x_3\overline{x_5}$
$f_{t_{\{1,3\}}} = x_1x_2x_3x_5 + \overline{x_2}x_4x_5 + x_2x_3x_4x_5 + x_3\overline{x_5}$

Since $f = f_{t_{\{1,3\}}}$, Theorem 2 ensures that $P_{t_3} = x_2$ and $P_{t_3} = x_5$ also makes $f = f_{t_{\{1,3\}}}$. Additionally, $f = f_{t_{\{1,3\}}} = f_{t_{\{1\}}} = f_{t_{\{3\}}}$. Note that our fault tolerance calculations consider all possible literal combinations of P_t's. As a result, all tolerable stuck-closed fault positions are found.

Fault tolerance performance FT_{sc} of an $N \times M$ crossbar can be calculated by using

$$FT_{sc} = \sum_{i=0}^{max\{AL\}} C_i(1 - p_{sc})^{Z-AL_i} p_{sc}^{AL_i}$$

where p_{sc} is an independent stuck-closed fault probability of each switch; C_i is the number of cases tolerating i faults; AL_i is the number of added literals to the function f representing the number of faulty switches; and $Z = N \cdot M \cdot (1 - IR)$. Note that $Z - AL_i$ represents the number of unused switches in crossbars. Note that C_0 represents a fault-free condition and always $C_0 = 1$. For Example 1, $N = 4$, $M = 7$, and $IR = 10/28$ that results in $Z = 18$. Additionally $C_1 = 3, C_2 = 3$ and $C_3 = 1$, and suppose that $p_{sc} = 2\%$. As a result, FT_{sc} is calculated as 74%.

Fault Tolerance for Multi-Output Functions

Although we develop our method for stuck-closed faults using single-output functions, we can directly apply it to multi-output functions. We only need a modification for the first step of our method, obtaining all P_{t_i}'s. First, we need to obtain all P_{t_i}'s for each output function separately. If a product is used by multiple outputs then only common P_{t_i}'s for this product are used. If a product is used by a single output then we use all of the corresponding P_{t_i}'s. After having P_{t_i}'s in the first step, we follow the same procedure as we do in the second step of our method developed for single-output functions. To elucidate our method for multi-output functions, we present an example.

Example 2: Considering target functions in ISOP form $f_1 = x_1x_2 + x_1\overline{x_3} + x_2\overline{x_4} + x_3x_5$ and $f_2 = x_1x_2 + x_1\overline{x_3} + \overline{x_2}\ \overline{x_4} + x_4x_5$. Implementation is shown in Figure 12. Literal set (LS) of f_1 and f_2 is LS = $\{x_1, x_2, x_3, x_4, x_5, \overline{x_2}, \overline{x_3}, \overline{x_4}\}$.

1.Step: We find faults affecting single products by exploiting Theorem 1. We only consider literals being member of LS.

For f_1:
$\overline{g_1(P_1 = 1)} = x_3x_4$
$P_{t_1} = x_3, P_{t_1} = x_4, P_{t_1} = x_3x_4$

$\overline{g_2(P_2 = 1)} = \overline{x_2}$
$P_{t_2} = \overline{x_2}$

$\overline{g_3(P_3 = 1)} = \overline{x_1}$
P_{t_3} : not a member of LS

$\overline{g_4(P_4 = 1)} = (\overline{x_1}x_4 + \overline{x_2})$
P_{t_4} : no single literal

For f_2:
$\overline{g_1(P_1 = 1)} = x_3(\overline{x_4} + \overline{x_5})$
$P_{t_1} = x_3$

$\overline{g_2(P_2 = 1)} = \overline{x_2}x_4\overline{x_5}$
$P_{t_2} = \overline{x_2}, P_{t_2} = x_4, P_{t_2} = \overline{x_2}x_4$

$\overline{g_3(P_3 = 1)} = (\overline{x_1} + x_3)$
P_{t_3} : no single literal

$\overline{g_4(P_4 = 1)} = (\overline{x_1} + \overline{x_2}x_3)$

Multi-Output Implementations

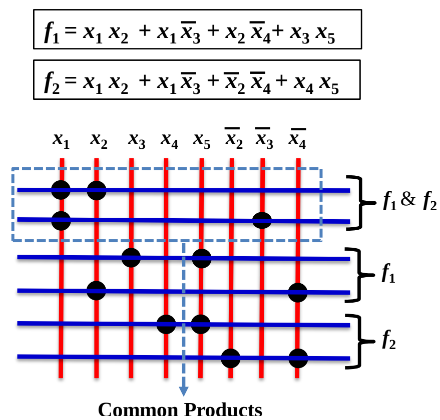

Fig. 12. A crossbar implementation in case of multi-outputs showing common products found in f_1 and f_2.

P_{t_4} : no single literal

Since P_1 and P_2 are common products, we should choose common P_t's for these products that are $P_{t_1} = x_3$ and $P_{t_2} = \overline{x_2}$, so the tolerance condition is met for both functions.

2.Step: We first check whether f_1 equals to $f_{1,t_{\{1,2\}}}$.
$P_{t_1} = x_3$
$P_{t_2} = \overline{x_2}$
$f_{1,t_{\{1,2\}}} = x_1x_2x_3 + x_1\overline{x_2}\ \overline{x_3} + x_2\ \overline{x_4} + x_3x_5$

Since $f_{1,t_{\{1,2\}}} \neq f_2$ and no more products left, we stop.

We check whether f_2 equals to $f_{2,t_{\{1,2\}}}$.
$P_{t_1} = x_3$
$P_{t_2} = \overline{x_2}$
$f_{2,t_{\{1,2\}}} = x_1x_2x_3 + x_1\overline{x_2}\ \overline{x_3} + \overline{x_2}\ \overline{x_4} + x_4x_5$

Since $f_{2,t_{\{1,2\}}} \neq f_2$ and no more products left, we stop.

For the above example, $N = 6$, $M = 8$, and $IR = 16/48$ that results in $Z = 32$. Additionally, $C_1 = 2$ and suppose that $p_{sc} = 2\%$. As a result, FT_{sc} is calculated as 54%.

C. Performance Evaluation

Our method finds all probable places of tolerable stuck-open and stuck-closed transient faults occurring in nano-crossbars. Using our method transient fault tolerance performances of the crossbars can be also calculated. As opposed to the methods using randomly assigned faults on crossbars such as a Monte Carlo method, our method purely uses algebraic equations to find fault performances. This allows to achieve accurate results even for considerably large crossbars.

Table III shows fault tolerance performances FT_{so} and FT_{sc} for few benchmark functions with a fault probability of 5%. For stuck-open faults, since it is not possible to tolerate faults occurring on used switches, the performance is directly calculated using the logic inclusion ratio and the crossbar size. However, for stuck-closed faults there are some cases such

TABLE III
PERFORMANCE OF BENCHMARK FUNCTIONS FOR TRANSIENT FAULTS WITH 5% FAULT RATE.

Circuit Name	Stuck-open	Stuck-closed	
		Direct results	Accurate results with the proposed method
B12 1	23%	16%	21%
B12 6	19%	14%	16%
B12 7	19%	14%	19%
C17 0	73%	73%	77%
Dc1 2	54%	44%	53%
Dc1 6	73%	63%	66%
Misex1 7	48%	32%	35%

that faults on unused switches are tolerated. Table III shows results derived by neglecting these cases (direct results) and by considering them via the proposed method (accurate results); there is as high as 9% difference between the values.

Our method is applicable to both single-output and multi-output functions as justified in the previous section. Another important consideration is redundancy. Although in this study, we suppose that target functions are implemented in irredundant sum-of-products (ISOP) forms to minimize the number of used switches for cost optimization in fabrication, this is not a necessary condition to apply our tolerance method. In case of having redundancy in literal level with addition of literals to products, by keeping the number of products same, our method is directly applicable to find all possible positions of tolerable faults in the crossbar. We only need to have an ISOP form of the given expression in SOP form. Indeed, adding a literal to a prime implicant is the base of our method for stuck-closed faults. Here, the difference comes in the calculation of fault tolerance performances FT_{so} and FT_{sc}; given formulas in the previous section need to be updated that would result in an increase and decrease in FT_{so} and FT_{sc} values, respectively.

In case of having redundancy in product level, having multiple lines/wires implementing the same product (as a prime implicant), our method can be directly applicable for stuck-open faults including the calculation of FT_{so} since removing any literal from a prime implicant results in a new function. However, for suck-closed faults we need modifications especially for Theorem 1. Here, if a product P_i is implemented A times then for each of the A wires, we need to calculate P_{t_i}'s by considering negated forms of products having at most A literals in $g_i(P_i = 1)$. The calculation of FT_{sc} should be also changed accordingly. One can also consider redundancy both in literal and product levels. Lets explain this with an example using different implementations with different redundancies:

Example 3: Consider a target function in ISOP form $f = x_1x_2x_3 + \overline{x_2}x_4x_5 + x_3x_4 + x_3\overline{x_5}$ that is the same function used in Example 1. Consider different implementations of f using different types of redundancies in Figure 13.

Figure 13 (a) shows an implementation of f with literal level redundancy by a 4×7 crossbar. Assume that we have a 5% stuck-open fault rate. Tolerable cases become no fault with $(1 - 0.05)^{12} = 54\%$ probability, single fault with $2 \times (1 - 0.05)^{11} 0.05^1 = 5\%$ probability, and two faults with $(1 - 0.05)^{10} 0.05^2 = 0.1\%$ probability. At the end $FT_{so} = 54\% + 5\% + 0.1\% \approx 60\%$. For stuck-closed faults,

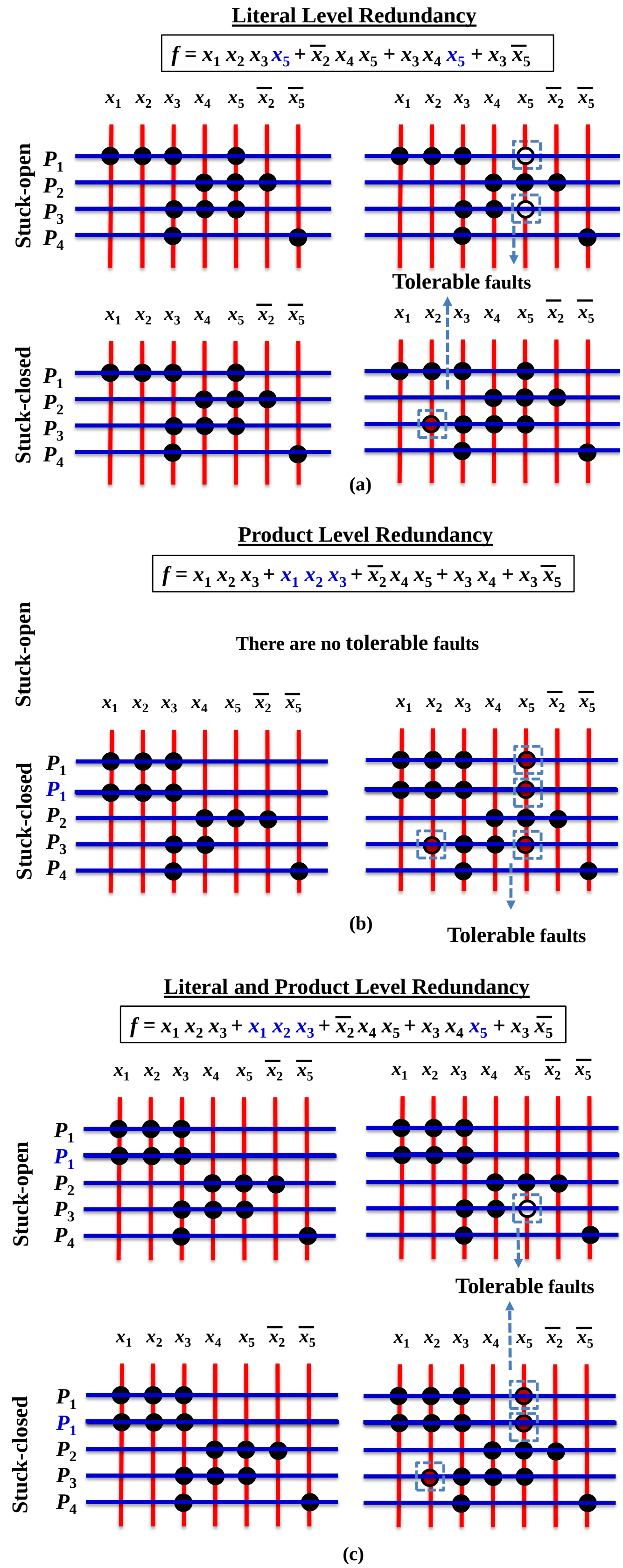

Fig. 13. Tolerance with redundancy based implementations (a) literal level redundancy (b) product level redundancy (c) literal and product level redundancy.

we already determine the tolerable positions in Example 1 as $P_{t_1} = x_5$, $P_{t_3} = x_2x_5$, and their literal combinations. It is shown in Figure 13 (a) that $P_{t_1} = x_5$ and $P_{t_3} = x_5$ are covered by literal redundancies, so only tolerable fault is $P_{t_3} = x_2$. In this case, $N = 4$ and $M = 7$ that results in $Z = 16$. Additionally $C_1 = 1$, and suppose that $p_{sc} = 2\%$. As a result, FT_{sc} is calculated as 73%.

Figure 13 (b) shows an implementation of f with product level redundancy by a 5×7 crossbar. Even though a redundant product is used, we are still working with prime implicants. So no literal can be erased from any product. Therefore, with a 5% stuck-open fault rate, FT_{so} becomes $(1-0.05)^{13} = 51\%$. For stuck-closed faults, it is shown in Figure 13 (b) that an extra tolerable fault x_5 comes from the product redundancy, so $P_{t_1} = x_5$, $P_{t_3} = x_2x_5$, and $P_{t_1} = x_5$. Calculating all literal combinations with $N = 5$ and $M = 7$ results in $Z = 22$. Additionally, $C_1 = 4, C_2 = 6$, $C_3 = 4$, and $C_4 = 1$. Also suppose that $p_{sc} = 2\%$. As a result, FT_{sc} is calculated as 69%.

Figure 13 (c) shows an implementation of f with with literal and product level redundancies by a 5×7 crossbar. Assume that we have a 5% stuck-open fault rate. Tolerable cases become no fault with $(1-0.05)^{14} = 48\%$ probability and single fault with $(1-0.05)^{13}0.05^{1} = 2\%$ probability. At the end, $FT_{so} = 48\% + 2\% = 50\%$. For stuck-closed faults, $P_{t_3} = x_5$ is covered by a literal redundancy, so $P_{t_1} = x_5$, $P_{t_1} = x_5$, and $P_{t_3} = x_2$. In this case, $N = 5$ and $M = 7$, that results in $Z = 21$. Additionally, $C_1 = 3$, $C_2 = 3$, and $C_3 = 1$. Also suppose that $p_{sc} = 2\%$. As a result, FT_{sc} is calculated as 69%.

IV. Experimental Results

In this section, we present experimental results for our algorithm dealing with permanent faults given in Section II. We use standard benchmark circuits to measure fault tolerance performances of nano-crossbars [28]. We mostly consider an independent fault probability/rate (**Pf**) of 15% for each crosspoint that is an accepted upper limit for nano-crossbars [27]. We also try higher fault rates to test our algorithm's performance limits. Simulations are conducted in MATLAB. Crossbars with random faults are produced with MATLAB's predetermined matrix generator; only stuck-open faults are considered for consistency. All experiments run on a 3.30GHz Intel Core i5 CPU (only single core used) with 4GB memory. All the benchmark functions used in the simulations and the source code of proposed algorithm with supporting material are available at http://www.ecc.itu.edu.tr/images/f/f2/Fault_Tolerant_Logic_Mapping_MATLAB.zip

A. Runtime, Success Rate, and Accuracy

For a given target function with a certain function matrix size, we consider crossbar matrices both in optimal row-column sizes and in 1.5 times larger sizes. Although optimal crossbar sizes are desired for area considerations, it is quite challenging to find a mapping and that is why using 1.5 larger sizes are preferred in the literature [16] [17] [18] [22]. The larger the crossbar, the easier to find a valid mapping due to an exponential increase in solution space regarding the number of probable permutations.

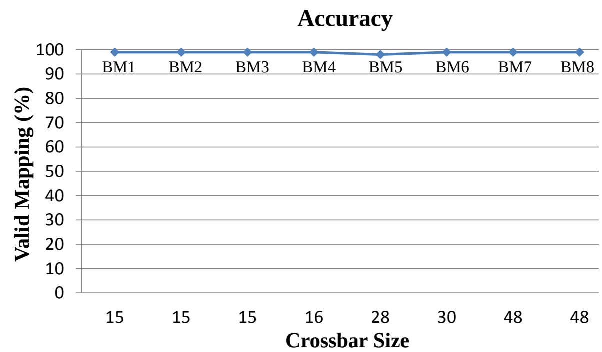

Fig. 14. Accuracy of the proposed algorithm for optimal size crossbars using 8 different benchmark circuits.

Table IV shows runtime and success rate values of the proposed algorithm for benchmark circuits with 15% stuck-open fault rate. We select a sample size of 600 around which average runtime and success rate (probability of success - **Psucc**) values become steady. Success rate is calculated as a ratio of the number of samples with valid mappings/matchings to the total sample size of 600. As seen from the table, our algorithm successfully finds mappings for considerably large benchmark circuits. To our knowledge no other algorithm is able to find a valid mapping for benchmarks "table5" and "t481". Examining the numbers in Table IV, we see that our algorithm does not need a permutation for 1.5 larger crossbars. We also see that although selecting 1.5 larger crossbars always reduces the runtime values, it does not necessarily result in better fault tolerance performances. Optimal size crossbars can also perfectly tolerate faults. To elaborate on this, we perform accuracy analysis as shown in Figure 14. We compare our optimal size mapping results with those of an exhaustive search algorithm. Since it is intractable to implement an exhaustive search for crossbar sizes larger than 7×7, only results pertaining to this limit are presented in Figure 14 that show an accuracy of at least 99% for 8 different benchmarks BM1 through BM8.

In Table V and Table VI, runtime comparisons of the memetic algorithm with fitness approximation (**MA/FA**) [22] and the proposed heuristic algorithm are given. We use the memetic algorithm since to our knowledge it is the fastest and the most efficient algorithm especially for large crossbars. We run the publicly posted code from [22] and tailor it for our benchmark functions which is not included in the referenced paper.

Examining the numbers in Table V and Table VI, we see that our runtime values are always better than those of the memetic algorithm. The memetic algorithm is not able to find a valid mapping for large functions such as *9sao*, *table5*, and *t481* under a reasonable time constraint. Additionally, while runtime values of the memetic algorithm for large benchmark circuits produce relatively high standard deviation, our runtimes are almost stable. Another aspect is that, the memetic algorithm is not as immune to an increase in fault rate as the proposed algorithm does.

TABLE IV
SUCCESS RATE (%), RUNTIME (S), AND AVERAGE PERMUTATION VALUES OF THE PROPOSED ALGORITHM FOR OPTIMAL AND 1.5 LARGER CROSSBAR SIZES WITH 15% STUCK-OPEN FAULT RATE.

Benchmark	Size	IR	Optimal Size			1.5 Larger Size		
			Psucc	Runtime(s)	Avg. Per.	Psucc	Runtime(s)	Avg. Per.
5xp1	75 x 14	28%	100%	0.001	0	100%	0.001	0
inc	34 x 14	40%	95%	0.29	450	100%	0.001	0
clip	167 x 18	29%	100%	0.032	4	100%	0.01	0
misex2	50 x 29	12%	100%	0.005	4	100%	0.002	0
9sym	87 x 18	33%	100%	0.008	1	100%	0.005	0
bw	65 x 10	35%	100%	0.01	4	100%	0.002	0
rd53	32 x 10	45%	100%	0.003	5	100%	0.001	0
rd73	141 x 14	42%	100%	0.13	18	100%	0.01	0
9sao	58 x 20	36%	0%	3.85	3000	100%	0.003	0
table5	158 x 34	36%	0%	27.7	3000	100%	0.02	0
t481	481 x 32	30%	0%	362.08	3000	100%	0.2	0

TABLE V
SUCCESS RATE (%) AND RUNTIME (SECOND) VALUES OF THE MEMETIC AND THE PROPOSED ALGORITHMS FOR 1.5 LARGER CROSSBAR SIZES WITH DIFFERENT STUCK-OPEN FAULT RATES.

Benchmark	Size	MA/FA [22]						Proposed Algorithm					
		Pf=15%		Pf=20%		Pf=30%		Pf=15%		Pf=20%		Pf=30%	
		Psucc	Time	Psucc	Time	Psucc	Time	Psucc	Time	Psucc	Time	Psucc	Time
5xp1	75 x 14	100%	0.702	-	-	-	-	100%	0.001	100%	0.003	100%	0.003
inc	34 x 14	100%	0.110	67%	14.93	-	-	100%	0.001	100%	0.007	100%	0.007
clip	167 x 18	100%	-	-	-	-	-	100%	0.01	100%	0.015	100%	0.020
misex2	50 x 29	100%	0.008	100%	0.354	100%	0.374	100%	0.002	100%	0.020	100%	0.028
9sym	87 x 18	100%	0.109	-	-	-	-	100%	0.005	100%	0.005	100%	0.007
bw	65 x 10	100%	0.798	-	-	-	-	100%	0.002	100%	0.001	100%	0.002
rd53	32 x 10	100%	0.074	100%	0.336	82%	12.67	100%	0.001	100%	0.001	100%	0.001
rd73	141 x 14	-	-	-	-	-	-	100%	0.01	100%	0.012	100%	0.021
9sao	58 x 20	-	-	-	-	-	-	100%	0.003	100%	0.003	0%	6.65
table5	158 x 34	-	-	-	-	-	-	100%	0.024	0%	51.38	0%	36.38
t481	481 x 32	-	-	-	-	-	-	100%	0.208	100%	0.303	0%	423.2

TABLE VI
SUCCESS RATE (%) AND RUNTIME (SECOND) COMPARISON OF THE MEMETIC AND THE PROPOSED ALGORITHMS FOR 16×16 AND 24×24 SIZE BENCHMARKS USING 1.5 LARGER CROSSBAR SIZES, STUCK-OPEN FAULT RATE: 15%, IR: 40%.

No	Size = 16 × 16				Size = 24 × 24			
	MA/FA [22]		Proposed Algorithm		MA/FA [22]		Proposed Algorithm	
	Psucc	Runtime(s)	Psucc	Runtime(s)	Psucc	Runtime(s)	Psucc	Runtime(s)
1	100%	0.004	100%	0.002	100%	0.006	100%	0.002
2	100%	0.002	100%	0.001	100%	0.005	100%	0.001
3	100%	0.002	100%	0.001	100%	0.004	100%	0.002
4	100%	0.004	100%	0.001	100%	0.005	100%	0.002
5	100%	0.007	100%	0.001	100%	0.004	100%	0.001
6	100%	0.003	100%	0.001	100%	0.004	100%	0.002
7	100%	0.003	100%	0.001	100%	0.004	100%	0.001
8	100%	0.003	100%	0.001	100%	0.005	100%	0.001
9	100%	0.004	100%	0.001	100%	0.005	100%	0.002
10	100%	0.003	100%	0.001	100%	0.005	100%	0.002
11	100%	0.002	100%	0.001	100%	0.006	100%	0.002
12	100%	0.007	100%	0.001	100%	0.005	100%	0.002
13	100%	0.004	100%	0.001	100%	0.005	100%	0.002
14	100%	0.007	100%	0.001	100%	0.004	100%	0.001
15	100%	0.002	100%	0.001	100%	0.005	100%	0.001
16	100%	0.003	100%	0.001	100%	0.007	100%	0.002
17	100%	0.008	100%	0.001	100%	0.005	100%	0.001
18	100%	0.003	100%	0.001	100%	0.004	100%	0.002
19	100%	0.002	100%	0.001	100%	0.007	100%	0.002
20	100%	0.002	100%	0.001	100%	0.004	100%	0.001

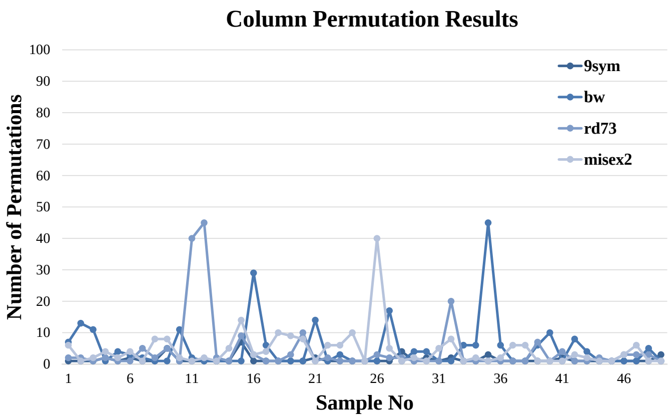

Fig. 15. Number of permutations to find a valid mapping for each sample using optimal size crossbars.

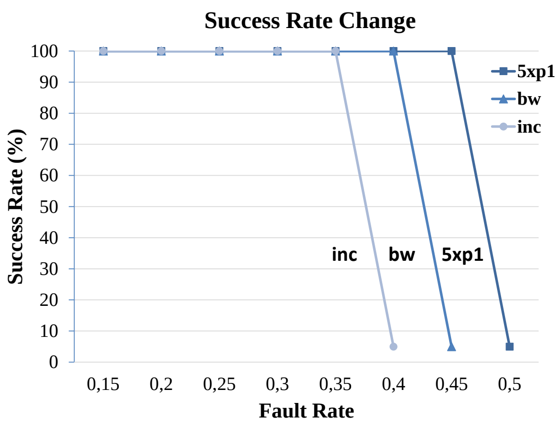

Fig. 16. Success rate versus fault rate; **inc**, **bw**, and **5xp1** have IR's of 40%, 35%, and 28%, respectively.

B. *Effectiveness and Limitations*

In our algorithm if no matching is found initially, column permutations are changed to find a matching that is repeated at most PL times. Experimentally we found that PL = 3000 for our benchmarks. The reason of selecting 3000 as a trial limit is our goal of maintaining minimum of 95% success rate. Indeed, for most cases repeating is not necessitated. Especially for 1.5 larger crossbar sizes, no permutation is needed at all; all results with having non zero success rates in Table IV, Table V, and Table VI do not need any a permutation ($PL = 0$). However, for optimal sizes, we sometimes need permutations; Figure 15 illustrates this by presenting the number of permutations for different benchmark circuits using 50 samples.

We explore our algorithm's performance limitations by increasing fault rates and row/column sizes. The limitations are directly correlated to the size of the solution space. As expected, the solution space diminishes if fault rates are getting close to IR and 1-IR in the presence of stuck-closed, and stuck-open faults, respectively. This is illustrated in Figure 16 for stuck-open faults using 1.5 times larger crossbars. Here, success rates drop sharply after certain threshold values that are positively correlated with 1-IR values of the benchmarks.

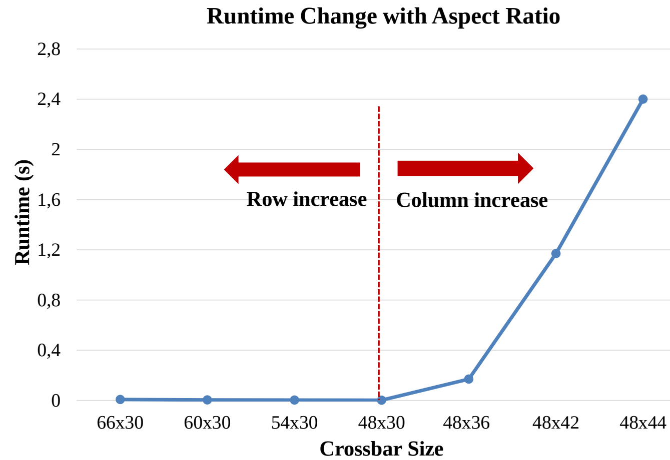

Fig. 17. Runtime changes with an increase in either row or column size, IR=40%.

Increasing row or column sizes also affect the solution space. Recall that our algorithm uses a constant permutation for one dimension (column) and advancing through the other one (row) that reduces the number of operations for finding a valid mapping. Therefore, while increasing row sizes does not directly affect the solution space for matchings, an increase in column size dramatically reduces it. To overcome this problem, our algorithm transposes given matrices to satisfy that the number of columns is always less than or equal to the number of rows. To see the effects of column and row increases to our algorithm, we discard transposing operation. The results are given in Figure 17 for stuck-open faults using 1.5 times larger crossbars and IR=0.4. As it appears from the figure, the runtime sharply increases from 0.002s to 1.2s if the crossbar size increases from 48×30 to 48×42. As a result, for the same size crossbars, same $N \cdot M$, our algorithm works more satisfactorily if the crossbar column and row sizes are more apart from each other.

Another limitation of our algorithm would be its accuracy in case of having a small solution space. Indeed, this is a general problem for heuristic algorithms. To overcome this problem, exact algorithms exploiting a sub-graph isomorphism can be used [29] if runtime is not a main concern. In addition, a slower algorithm using pruning techniques can be exploited [30].

V. Conclusion

In this study, we propose a fast heuristic algorithm to tolerate permanent faults in nano-crossbar arrays by exploiting the techniques of index sorting, backtracking, and row matching. The algorithm's effectiveness is demonstrated on standard benchmark circuits in comparison with the related studies in the literature. Also we develop a method to accurately analyse transient fault tolerance of nano-crossbar arrays. The method formally and recursively finds tolerable fault positions represented by Boolean logic expressions. Using the method, transient fault tolerance performances of the crossbars can be calculated.

Throughout this study, we treat stuck-closed and stuck-open faults separately. Indeed, for permanent faults our algorithm works properly in case having both fault types in crossbars. Matrices are sorted according to stuck-closed and stuck-open faults in case of having a higher stuck-closed and stuck-open fault rates, respectively. However, the efficiency of the algorithm would not be satisfactory if we have close fault rates. This is considered as a future work. Another future direction is to develop circuit design and optimization techniques for given fault tolerance specifications by simultaneously treating permanent and transient faults. We also aim to extend this study to be applicable for different emerging technologies including magnetic, memristive, and organic switch based nanoarrays.

References

[1] O. Tunali and M. Altun, "Defect tolerance in diode, fet, and four-terminal switch based nano-crossbar arrays," in *Nanoscale Architectures (NANOARCH), 2015 IEEE/ACM International Symposium on*. IEEE, 2015, pp. 82–87.

[2] H. Yan, H. S. Choe, S. Nam, Y. Hu, S. Das, J. F. Klemic, J. C. Ellenbogen, and C. M. Lieber, "Programmable nanowire circuits for nanoprocessors," *Nature*, vol. 470, no. 7333, pp. 240–244, 2011.

[3] J. Yao, H. Yan, S. Das, J. F. Klemic, J. C. Ellenbogen, and C. M. Lieber, "Nanowire nanocomputer as a finite-state machine," *Proceedings of the National Academy of Sciences*, vol. 111, no. 7, pp. 2431–2435, 2014.

[4] H. Hamoudi, "Crossbar nanoarchitectonics of the crosslinked self-assembled monolayer," *Nanoscale research letters*, vol. 9, no. 1, pp. 1–7, 2014.

[5] Y. Chen, G.-Y. Jung, D. A. Ohlberg, X. Li, D. R. Stewart, J. O. Jeppesen, K. A. Nielsen, J. F. Stoddart, and R. S. Williams, "Nanoscale molecular-switch crossbar circuits," *Nanotechnology*, vol. 14, no. 4, p. 462, 2003.

[6] M. Gholipour and N. Masoumi, "Design investigation of nanoelectronic circuits using crossbar-based nanoarchitectures," *Microelectronics Journal*, vol. 44, no. 3, pp. 190–200, 2013.

[7] G. Snider, P. Kuekes, and R. S. Williams, "Cmos-like logic in defective, nanoscale crossbars," *Nanotechnology*, vol. 15, no. 8, p. 881, 2004.

[8] C. Collier, E. Wong, M. Belohradskỳ, F. Raymo, J. Stoddart, P. Kuekes, R. Williams, and J. Heath, "Electronically configurable molecular-based logic gates," *Science*, vol. 285, no. 5426, pp. 391–394, 1999.

[9] A. DeHon and B. Gojman, "Crystals and snowflakes: building computation from nanowire crossbars," *Computer*, no. 2, pp. 37–45, 2011.

[10] J. Huang, M. B. Tahoori, and F. Lombardi, "On the defect tolerance of nano-scale two-dimensional crossbars," in *Defect and Fault Tolerance in VLSI Systems, 2004. DFT 2004. Proceedings. 19th IEEE International Symposium on*. IEEE, 2004, pp. 96–104.

[11] W. Feng, F. Lombardi, H. A. Almurib, and T. N. Kumar, "Testing a nanocrossbar for multiple fault detection," *Nanotechnology, IEEE Transactions on*, vol. 12, no. 4, pp. 477–485, 2013.

[12] A. M. S. Shrestha, S. Tayu, and S. Ueno, "Orthogonal ray graphs and nano-pla design." in *ISCAS*, 2009, pp. 2930–2933.

[13] M. B. Tahoori, "A mapping algorithm for defect-tolerance of reconfigurable nano-architectures," in *Computer-Aided Design, 2005. ICCAD-2005. IEEE/ACM International Conference on*. IEEE, 2005, pp. 668–672.

[14] A. Al-Yamani, S. Ramsundar, D. K. Pradhan *et al.*, "A defect tolerance scheme for nanotechnology circuits," *Circuits and Systems I: Regular Papers, IEEE Transactions on*, vol. 54, no. 11, pp. 2402–2409, 2007.

[15] B. Yuan and B. Li, "A fast extraction algorithm for defect-free subcrossbar in nanoelectronic crossbar," *ACM Journal on Emerging Technologies in Computing Systems (JETC)*, vol. 10, no. 3, p. 25, 2014.

[16] S. Gören, H. F. Ugurdag, and O. Palaz, "Defect-aware nanocrossbar logic mapping through matrix canonization using two-dimensional radix sort," *ACM Journal on Emerging Technologies in Computing Systems (JETC)*, vol. 7, no. 3, p. 12, 2011.

[17] Y. Su and W. Rao, "An integrated framework toward defect-tolerant logic implementation onto nanocrossbars," *Computer-Aided Design of Integrated Circuits and Systems, IEEE Transactions on*, vol. 33, no. 1, pp. 64–75, 2014.

[18] M. Zamani, H. Mirzaei, and M. B. Tahoori, "Ilp formulations for variation/defect-tolerant logic mapping on crossbar nano-architectures," *ACM Journal on Emerging Technologies in Computing Systems (JETC)*, vol. 9, no. 3, p. 21, 2013.

[19] W. Rao, A. Orailoglu, and R. Karri, "Topology aware mapping of logic functions onto nanowire-based crossbar architectures," in *Proceedings of the 43rd annual Design Automation Conference*. ACM, 2006, pp. 723–726.

[20] A. DeHon and H. Naeimi, "Seven strategies for tolerating highly defective fabrication," *Design & Test of Computers, IEEE*, vol. 22, no. 4, pp. 306–315, 2005.

[21] J.-S. Yang and R. Datta, "Efficient function mapping in nanoscale crossbar architecture," in *Defect and Fault Tolerance in VLSI and Nanotechnology Systems (DFT), 2011 IEEE International Symposium on*. IEEE, 2011, pp. 190–196.

[22] B. Yuan, B. Li, T. Weise, and X. Yao, "A new memetic algorithm with fitness approximation for the defect-tolerant logic mapping in crossbar-based nanoarchitectures," *Evolutionary Computation, IEEE Transactions on*, vol. 18, no. 6, pp. 846–859, 2014.

[23] M. O. Simsir, S. Cadambi, F. Ivančić, M. Roetteler, and N. K. Jha, "A hybrid nano-cmos architecture for defect and fault tolerance," *ACM Journal on Emerging Technologies in Computing Systems (JETC)*, vol. 5, no. 3, p. 14, 2009.

[24] I. Polian and W. Rao, "Selective hardening of nanopla circuits," in *Defect and Fault Tolerance of VLSI Systems, 2008. DFTVS'08. IEEE International Symposium on*. IEEE, 2008, pp. 263–271.

[25] W. Rao, A. Orailoglu, and R. Karri, "Logic level fault tolerance approaches targeting nanoelectronics plas," in *Design, Automation & Test in Europe Conference & Exhibition, 2007. DATE'07*. IEEE, 2007, pp. 1–5.

[26] S. Baranov, I. Levin, O. Keren, and M. Karpovsky, "Designing fault tolerant fsm by nano-pla," in *On-Line Testing Symposium, 2009. IOLTS 2009. 15th IEEE International*. IEEE, 2009, pp. 229–234.

[27] M. Haselman and S. Hauck, "The future of integrated circuits: A survey of nanoelectronics," *Proceedings of the IEEE*, vol. 98, no. 1, pp. 11–38, 2010.

[28] K. McElvain, "Iwls'93 benchmark set: Version 4.0," in *Distributed as part of the MCNC International Workshop on Logic Synthesis*, vol. 93, 1993.

[29] A. Aho, J. Hopcroft, and J. Ullman, "Computers and intractability: A guide to np-completeness," 1979.

[30] L. P. Cordella, P. Foggia, C. Sansone, and M. Vento, "A (sub) graph isomorphism algorithm for matching large graphs," *Pattern Analysis and Machine Intelligence, IEEE Transactions on*, vol. 26, no. 10, pp. 1367–1372, 2004.