A Fast Logic Mapping Algorithm for Multiple-type-Defect Tolerance in Reconfigurable Nano-Crossbar Arrays

Onur Tunalı and Mustafa Altun

Abstract—Unlike conventional CMOS circuits, nano-crossbar arrays have considerably high defect rates. Multiple-type defects randomly occur both on crosspoint switches and wires that substantially complicates the design phase of the circuits with an elimination of systematic design choices. In order to overcome this problem, a logic mapping methodology is presented in this paper. A fast heuristic algorithm using pre-mapping logic morphing, defect oriented adaptive sorting, matching with Hadamard multiplication, and backtracking is introduced. The proposed algorithm covers both crosspoint defects including stuck-open and stuck-closed types and wire defects including bridging and broken types. Effects of stuck-closed defects, mostly disregarded in the literature, are studied in depth. In simulations, an industrial benchmark suit is used for obtaining runtime and success rate values of the proposed algorithm in comparison with those of the existing algorithms in the literature. A relative accuracy evaluation is also given in comparison with exact mapping techniques. Finally, the steps of the algorithm that are based on pre-mapping and heuristic matching techniques, are separately justified with experimental results.

Index Terms—Reconfigurable Nano-crossbars; Defect Tolerance; Switching Arrays.

◆

1 INTRODUCTION

Rapid developments in nanotechnology have made it possible to produce viable solutions to longstanding integration and miniaturization issues of electronic circuits [1], [2] and [3]. Single components such as a diode and a FET have been successfully built with carbon nanotubes or silicon nanowires [3]. More importantly, these realizations lead to programmable circuit architectures based on nano-crossbar arrays which operate similarly to conventional programmable logic arrays (PLA's) [4], [5], [6], and [7]. Two fully operational implementations as a nanoprocessor and a finite-state machine are shown to be feasible in [8] and [9].

A nano-crossbar is constructed from two layers of orthogonal wires/lines. Every crosspoint/junction acts as a switching element analogue to a diode or a FET in customary circuits. A diagram of a nano-crossbar is displayed in Figure 1 (a). Nano-crossbars are dominantly produced with bottom-up fabrication techniques that generates uniform and dense structures. However, higher density comes with a price: higher defect rates. Predicted maximum defect rates likely to present in end products deviate between 15% and 20% [10] and [11]. Additionally, post-fabrication defect rates of components are given as 16% and 7% in [8] and [9].

Since current CMOS based paradigm has very small defect rates in contrast to the nano-crossbars, it is not possible to practice prevalent design and manufacturing techniques. They are not capable of handling high defect rates, especially for logic mapping processes during the design phase. Logic mapping of a nano-crossbar is to implement a given logic function by assigning function elements to inputs and outputs of a circuit. In conventional architectures, logic mapping is a straightforward process due to uniform and defect-free structures. However, in nano-crossbars high defect rates substantially complicate the mapping process. Every manufactured crossbar needs to be adjusted individually with respect to a defect map showing the position of damaged elements. Motivated by this, we offer an efficient defect tolerant logic mapping method for nano-crossbars having diverse defect types such as stuck-closed, stuck-open, wire breakdown, and wire bridging. These types are illustrated in Figure 1 (b).

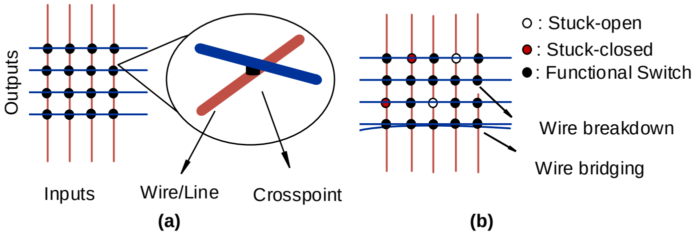


Fig. 1. Nano-crossbar array with (a) switching crosspoints (b) possible switch and wire defects.

In our mapping methodology, we do not need to comply with the constraints imposed by multi-level logic design which limits the input and output assignments and orderings. Two-level logic synthesis including AND and OR planes, is used for nano-crossbars, similar to PLA's. Furthermore, we only consider the configuration of the AND plane for defect tolerant logic mapping that is a common practice in the literature. The reason is that AND planes

---

- *This work is supported by the EU-H2020-RISE project NANOxCOMP #691178 and the TUBITAK-Career project #113E760.*
- *Onur Tunali and Mustafa Altun are with the Department of Nanoscience and Nanoengineering and Department of Electronics and Communication Engineering Istanbul Technical University, Istanbul, 34469.*
- *E-mails: onur.tunali, altunmus @ itu.edu.tr*

are generally much larger than OR planes; using a reconfigurability feature, a single line/wire as an OR plane is even sufficient to have every output a time, and correspondingly OR planes can be fabricated in a much more robust fashion with an aim of having defect-free crosspoints as suggested in [6]. Indeed, our mapping technique is fully applicable even for multi-level logic design if we have defect-free OR planes that allows us to select any ordering for AND planes without a constraint.

1.1 Previous Works

Our method is defect-aware meaning that it employs defective elements in the mapping process that allows us to use the crossbar area efficiently. On the other hand, defect-unaware methods aim to find the largest possible defect-free sub-crossbar from a defective full-crossbar that results in much lower yields especially for high defect rates [12] and [13]. As follows, we elaborate on defect-aware methods.

A common practice for the mapping problem is to use graph based models. Assignment algorithms including the Hungarian method [14] exploit a bipartite graph based model. Graph embedding and maximum bipartite graph matching are used in [15] and [16], respectively. Another algorithm uses graph canonization with sorting logic and crossbar adjacency matrices for the mapping [17]. Different from these methods, we use adaptive sorting by considering the dominant defect type. Additionally, we use a backtracking process for the mapping that eliminates overheads of initial graph construction and update that are severe especially for having multiple defect types.

A different method based on logic function manipulation is presented in [18]. It uses logic hardening and logic morphing methods during mapping to discard possible mismatches. Logic hardening benefits from redundancies by adding extra lines/wires; same function literals (variables and their negations) are assigned to multiple lines. This surely increases the chance of matching since having a single valid matching for each literal line is sufficient for valid mapping. Nevertheless, its impact on yield is overlooked; used crossbar sizes are two or three times larger than the sizes of the required defect-free crossbars. In the experimental results, we show that using the same crossbar sizes, it is possible to find a valid mapping without employing the proposed hardening method unless defect rates reach very high impractical values. In the same study, logic morphing is utilized during mapping which puts burden on the algorithm in each and every row/column matching process. We propose a pre-mapping morphing methodology that generates a logic function fine-tuned by considering given defect rates and can be used in all crossbars having the same or approximate defect rates. Further analysis of both methods are given in the experimental results.

Another approach is modelling the mapping problem as a Boolean satisfiability problem (SAT) [19]. Using SAT is truly inefficient especially for large crossbar sizes. In [17], both graph based and SAT based algorithms are implemented and it is observed that runtimes of the SAT based algorithm are always larger, close to one to two order of magnitude. The same drawback is visible in integer linear programming based models proposed in [20] and [21].

Apart from all these mentioned issues, underestimating the effect of stuck-closed defects is a general tendency in the literature. In [22] and [23], stuck-closed defects are entirely disregarded in the mapping process. In [24], [16], and [25], proposed algorithms are in compatible with stuck-closed defects, but no experimental result is given. In [18] and [17], although stuck-closed defects are considered in experimental results, chosen defect rates are rather low that prevents a comprehensive evaluation. In these studies, low stuck-closed defect rates are justified by projecting the underlying physical structure of nano-crossbars that is apparently not based on experimental data. Furthermore, all implementations require less than half of the switches in crossbars which makes the problem positively biased towards stuck-open defect types. For this reason, proposed methods concerning only stuck-open or low stuck-closed defect rates inherently gain an unfair leverage. In this paper, we propose an adaptive algorithm by equally considering stuck-closed and stuck-open defects with appointing a dominant character to a chosen defect type.

1.2 Overview and Contributions

Brief summary of this study and main contributions are as follows.

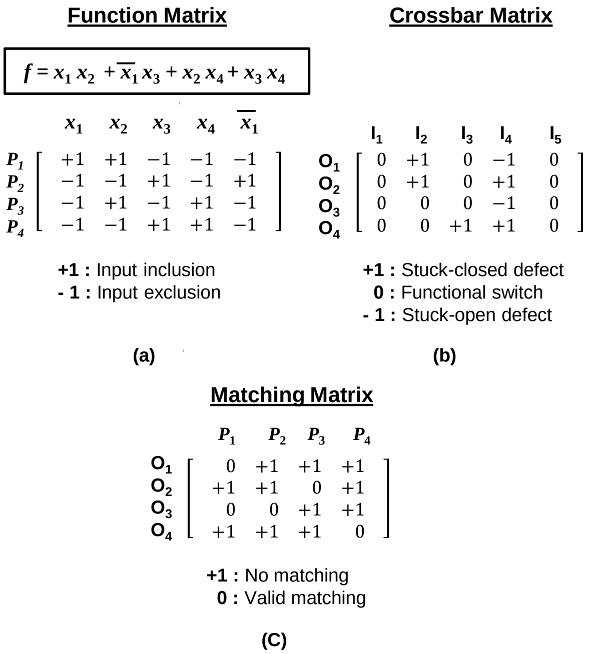
- We propose a defect-tolerant logic mapping algorithm for reconfigurable nano-crossbar arrays that uses pre-mapping logic morphing, sorting, and backtracking methods.
- We consider multiple-type defects by fine-tuning according to the defect type.
- We cover defect rates up to 20% in order to design a methodology resilient enough for potential fluctuations in the estimated defect rates.
- Our algorithm's success yield is 100% in most simulation cases; a failure of our algorithm is a strong indicator that even with exact algorithms finding a valid assignment is a rather demanding task.
- Our mapping algorithm works considerably faster than the existing algorithms in the literature.

The rest of the paper is organized as follows. In Section 2, we define key concepts and techniques. In Section 3, we present our algorithm that uses pre-mapping morphing, sorting, and backtracking methods. In Section 4, we present experimental results and elaborate on them. In Section 5, we summarize our contributions and discuss future directions.

2 PRELIMINARIES

In this section, we explain key concepts and models used in this study.

1) *Defect types* can be categorized under two main headings: switch and wire/line defects.
    a) *Switch defects* can be either stuck-closed or stuck-open. A stuck-closed defect makes a crosspoint switch permanently ON, so it can be considered as a logic 1. A stuck-open defect makes a crosspoint switch permanently OFF, so it can be considered as a logic 0.



$$f = x_1 x_2 + \overline{x}_1 x_3 + x_2 x_4 + x_3 x_4$$

$$\begin{array}{c} \\ P_1 \\ P_2 \\ P_3 \\ P_4 \end{array}\begin{array}{c} \begin{array}{ccccc} x_1 & x_2 & x_3 & x_4 & \overline{x_1} \end{array} \\ \begin{bmatrix} +1 & +1 & -1 & -1 & -1 \\ -1 & -1 & +1 & -1 & +1 \\ -1 & +1 & -1 & +1 & -1 \\ -1 & -1 & +1 & +1 & -1 \end{bmatrix} \end{array} \qquad \begin{array}{c} \\ O_1 \\ O_2 \\ O_3 \\ O_4 \end{array}\begin{array}{c} \begin{array}{ccccc} I_1 & I_2 & I_3 & I_4 & I_5 \end{array} \\ \begin{bmatrix} 0 & +1 & 0 & -1 & 0 \\ 0 & +1 & 0 & +1 & 0 \\ 0 & 0 & 0 & -1 & 0 \\ 0 & 0 & +1 & +1 & 0 \end{bmatrix} \end{array}$$

$$\begin{array}{c} \\ O_1 \\ O_2 \\ O_3 \\ O_4 \end{array}\begin{array}{c} \begin{array}{cccc} P_1 & P_2 & P_3 & P_4 \end{array} \\ \begin{bmatrix} 0 & +1 & +1 & +1 \\ +1 & +1 & 0 & +1 \\ 0 & 0 & +1 & +1 \\ +1 & +1 & +1 & 0 \end{bmatrix} \end{array}$$

Fig. 2. Matrix representations of (a) a logic function (b) a defective nanocrossbar (c) a matching matrix showing matching possibilities of function and crossbar matrices.

   b) *Wire defects* have two types that are breakdown and bridging wires. A breakdown defect makes the entire wire unusable, so the corresponding input/output should be excluded from the mapping process. A bridging defect makes two adjacent wires unusable, so the corresponding pair of inputs/outputs should be excluded from the mapping process as well.

2) *Function matrix* (FM) is a representation of a logic function in sum-of-products form such that the function's literals (variables and their negations) 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 an FM.

3) *Crossbar matrix* (CM) is a representation of a nanocrossbar such that its elements show either defective or functional switches. Figure 2 (b) shows a CM that can be also referred as a defect map.

   a) *Functional switches* are denoted with 0's that can be matched with +1's and -1's in an FM.
   b) *Stuck-closed switches* are denoted with +1's that can only be matched with +1's in an FM. A mismatched stuck-closed defect results in an addition of a literal to the corresponding product of a given function.
   c) *Stuck-open switches* are denoted with -1's that can only be matched with -1's in an FM. A mismatched stuck-open defect results in a removal of a literal from the corresponding product of a given function.

4) *Row matching with Hadamard product* starts with an element-by-element multiplication, called as *Hadamard multiplication* in this paper, similar to an inner product operation used for vectors. If there is a negative element of -1 in the resulting matrix then it means that there is no matching; otherwise there is a valid matching. Note that functional switches (denoted with 0's) in a CM can be always matched with +1's or -1's in an FM. However, +1's and -1's in a CM can only be matched with +1's and -1's in an FM, respectively. This is illustrated in Table 1. Additionally, Figure 3 shows an example for a valid matching between two rows of the matrices in case of having stuck-closed and stuck-open defects. This concept is proposed in [25].

TABLE 1
Element Compatibility of a Function Matrix (FM) and a Crossbar Matrix (CM)

FM_{ik}	CM_{ik}	$\mathrm{FM}_{ik} \times \mathrm{CM}_{ik}$	Matching
+1	+1	+1	Yes
+1	0	0	Yes
-1	0	0	Yes
-1	-1	+1	Yes
+1	-1	-1	No
-1	+1	-1	No

5) *Matching matrix* is constructed after performing row matchings of function and crossbar matrices. This is similar to a cost matrix used in assignment problems having an objective of minimizing the total cost. Figure 2(c) shows a matching matrix of function and crossbar matrices in Figure 2(a) and (b), respectively. A 0 and +1 elements of the matrix respectively show that a matching is possible and there is no matching. In other words, assignments yielding zero costs produce a valid mapping. Note that we only use the matching matrix in the implementation of an exact algorithm; our proposed algorithm does not need it.

6) *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 an FM. As an example, consider an FM in Figure 2 (a). Here, the number of +1's or the number of used switches is 8, so IR = 8 /20.

7) *Dominant defect type* is found by comparing the ratios of the stuck-closed defect rate to IR and the stuck-open defect rate to (1 - IR). The defect type with the larger ratio is appointed as dominant for which it is more difficult to find a matching.

8) *Dominant column index* is a 2-tuple (a,b) denoting the number +1's and -1's present in a column. The first element of the tuple (a) represents the dominant defect type, and sorting of indices is performed primarily for this element. For example, assuming a matrix having the following column indices: Column1 - (4,1), Column2 - (4,4), Column3 - (3,3), Column4 - (3,5), and Column5 - (2,4). If we sort indices in descending order then the result is Column2 - (4,4), Column1 - (4,1), Column4 - (3,5), Column3 - (3,3), and Column5 - (2,4).

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TETC.2017.2755458, IEEE Transactions on Emerging Topics in Computing

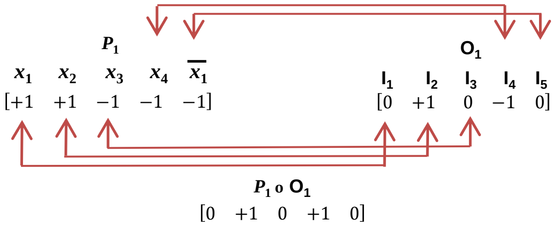

P_1
x_1 x_2 x_3 x_4 $\overline{x_1}$
[+1 +1 −1 −1 −1]
O_1
I_1 I_2 I_3 I_4 I_5
[0 +1 0 −1 0]
$P_1 \circ O_1$
[0 +1 0 +1 0]

Fig. 3. Matching two rows with Hadamard multiplication.

Algorithm 1 Heuristic Algorithm

```
Input: FM_{m×n} and CM_{m×n} $p_o, p_c$
Output: IA and OA                                   ▷ input and output arrays

FM = PRE-MAPPING_MORPHING($IRvalues$, FM_{m×n}, $f$, IR_{thr})
    ▷ Algorithm 2 is called

if m < n then
    FM = FM$^T$                                     ▷ transpose of the matrices
    CM = CM$^T$
end if

FM = FUNCTION_SORT(FM)
IA ← column_permutation of the FM
CM = CROSSBAR_SORT(CM)
OA = []     ▷ initially no assignment is made to output array

for k=1 to m do
    matching = false;
    F_k ← kth row of the FM
    OA = BACKTRACK_UNMATCHED(F_k, OA)        ▷ output array is updated with backtracking
    if ¬ matching then
        OA = BACKTRACK_MATCHED(F_k, OA)
    end if
end for

function FUNCTION_SORT(FM)
    $\pi_{row}$ ← Row permutation sorted according to the dominant defect type
    $\pi_{column}$ ← Column permutation sorted according to the dominant defect type
    FM← FM[$\pi_{row}, \pi_{column}$ ]
end function
function CROSSBAR_SORT(CM)
    $\pi_{row}$ ← Row permutation sorted according to the number of total defects
    $\pi_{column}$ ← Column permutation sorted according to the dominant column indices
    CM← CM[$\pi_{row}, \pi_{column}$ ]
end function

function BACKTRACK_UNMATCHED(F_k, OA)
    for j=1 to m and OA[j] $= \emptyset$ do
        C_j ← jth row of the CM
        if F_k .* C_j $\geq 0$ then          ▷ Hadamard multiplication
            OA[j] = k                         ▷ assigning process
            matching = true;
            break
        end if
    end for
end function

function BACKTRACK_MATCHED(F_k, OA)
    for j=1 to m and OA[j] $\neq \emptyset$ do
        C_j ← jth row of the CM
        if F_k .* C_j $\geq 0$ then          ▷ Hadamard multiplication
            F_OA[j] ← OA[j]th row of the FM
            OA = BACKTRACK_UNMATCHHED(F_OA[j], OA)
            if matching then
                OA[j] = k                     ▷ assigning process
                break
            end if
        end if
    end for
end function
```

3 ALGORITHM

One can consider the mapping problem as an assignment problem. To find a valid mapping, literals and products of a logic function should be appointed to inputs and outputs of a nano-crossbar yielding a correct assignment. Consider assignments of n literals (x's and their negations) and m products (P's) to the columns and rows of an FM, respectively. Also consider a CM with n input columns and m output rows. An input array IA and an output array OA are defined such that their i^{th} and j^{th}elements IA[i], $1 \leq i \leq n$, and OA[j], $1 \leq j \leq m$, are the assigned literal and product to the i^{th} crossbar input and the j^{th} crossbar output, respectively. As an example, if IA[2] = x_3 and OA[4] = P_1 then it means that x_3 and P_1 are assigned to the 2nd input and the 4th output of the crossbar, respectively. The proposed algorithm results in input and output arrays that establish a valid mapping or a correct assignment without any mismatches. A pseudocode of the algorithm is given in Algorithm 1.

The first step of the algorithm aims to improve stuck-closed defect tolerance. In this step, we perform pre-mapping morphing by adding redundant literals to the products of a given logic function. Thus, the first step only deals with the FM without changing its size.

The second step of the algorithm performs sorting both for the FM and the CM to make them ready for matching. Rows and columns of the FM are both sorted according to the number of dominant defects. On the other hand, rows and columns of the CM are sorted according to the number of total defects and the dominant column indices, respectively. The final sorted form of the literals (columns of the FM) are directly assigned to the IA.

Third and the final step of the algorithm involves row by row matching with Hadamard multiplication, and backtracking in case of having any mismatches. As a result the OA is constructed. Detailed explanations of these three steps are given in the following three sub-sections. Additionally, a performance evaluation of the algorithm is given in the last sub-section.

3.1 Pre-mapping Logic Morphing

Logic morphing considers equivalent forms of a logic function that can be exploited to tolerate defects. Considering that a stuck-closed defect adds a literal to the corresponding product and a stuck-open defect removes it, defect tolerance is possible if these additions and removals result in a same Boolean function as the one initially given. Figure 4 shows an example. Suppose that $f = x_1x_2 + x_1\overline{x_2}x_4 + x_1x_3$ is given and $f' = x_1x_2 + x_1x_4 + x_1\overline{x_2}x_3$ is obtained after having

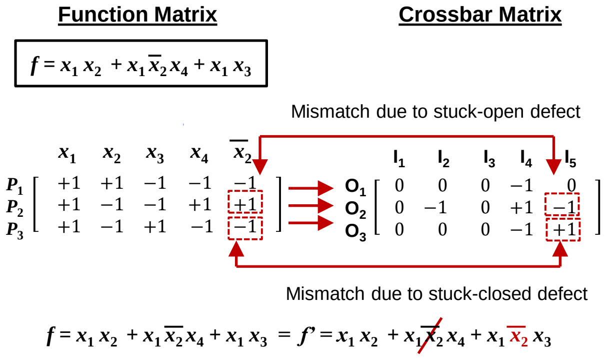

Function Matrix
Crossbar Matrix
f = x1 x2 + x1 x̄2 x4 + x1 x3
Mismatch due to stuck-open defect
x1 x2 x3 x4 x̄2
P1 +1 +1 −1 −1 −1
P2 +1 −1 −1 +1 +1
P3 +1 −1 +1 −1 −1
O1 O2 O3
I1 I2 I3 I4 I5
0 0 0 −1 0
0 −1 0 +1 −1
0 0 0 −1 +1
Mismatch due to stuck-closed defect
f = x1 x2 + x1 x̄2 x4 + x1 x3 = f' = x1 x2 + x1 x̄2 x4 + x1 x̄2 x3

Fig. 4. Mismatches due to stuck-closed and stuck-open defects. Although there are two mismatches, the resulted function $f^{'}$ equals to a given function f.

defects. Here, a stuck-closed defect results in an addition of $\overline{x_2}$, and a stuck-open defect results in a removal of $\overline{x_2}$. Since the resulted function $f^{'}$ is still equal the given function f, tolerance is achieved.

Our logic morphing technique aims to improve defect tolerance of only stuck-closed defects by adding redundant literals to products. Our motivation is based on two observations. First, function matrices always have IR values less than 50%, so stuck-open defects are much more likely to be tolerated. Note that to implement a logic function, a nano-crossbar uses literals as input lines. Since rows in an FM represent products of a logic function, maximum IR value of a row is 50% because both a variable and its negation can not be present in a product. Second observation is that logic functions in benchmarks are mostly given in minimal forms, so there is very limited chance of finding a redundant literal to be removed. Experimental results given in Section 4 also support our claims.

We apply our morphing technique before the mapping process, so it is a one-time operation. Additionally, we do not apply morphing to all products/rows. Adding redundant literals surely improves stuck-closed defect tolerance, but at the same time it makes harder to tolerate stuck-open defects. To determine which rows needing morphing, we define IR$_{\text{thr}}$, a threshold value for IR. If an IR value of a row is under IR$_{\text{thr}}$, we apply morphing; otherwise we do not apply. Indeed, IR$_{\text{thr}}$ corresponds to a case in which the matching probability of an FM row (p_m) is maximum. Here, the best-case scenario for mapping is IR=IR$_{\text{thr}}$ for each of the rows of the FM, and our purpose is getting close to this best-case by logic morphing.

Consider an FM with n inputs. Suppose that a row of the FM has X number of +1's where $1 \leq X \leq \frac{n}{2}$. Additionally, p_o, p_c, and p_f represent rates of stuck-open defects, stuck-closed defects, and functional switches, respectively. Note that $(p_o + p_c + p_f) = 1$. As a result, we derive the following equations:

$$p_m(X) = \sum_{k=0}^{X} \sum_{t=0}^{n-X} \frac{n!}{k!\,(n-k-t)!\,t!}\, p_c^k p_f^{n-k-t} p_o^t \tag{1}$$

$$max\Big(p_m(X=1), p_m(X=2), ..., p_m(X=X_{\text{thr}}), ..., p_m(X=\frac{n}{2})\Big) = p_m(X=X_{\text{thr}}) \tag{2}$$

$$\text{IR}_{\text{thr}} = \frac{X_{\text{thr}}}{n} \tag{3}$$

Equation (1) is used to find the matching probability of a row considering the all permutations. In deriving p_m, first we find the matching probability of a row with $p_c^k p_f^{n-k-t} p_o^t$ considering the functional and defective elements. Then we consider all different permutations of the row elements to make them matched that is in compatible with our mapping algorithm that performs sorting for this purpose. If a row of the CM has smaller number of +1's and -1's than a row of the FM does, then these two rows can be matched by applying proper permutations or sortings. Note that 0's in a CM row can be matched both with +1's and -1's in an FM row. Upper limits X and $n-X$ in Equation (1) ensure that this matching condition is met. The coefficient $\frac{n!}{k!\ (n-k-t)!\ t!}$ in Equation (1) considers different arrangements of a CM row respectively having $k, (n-k-t)$, and t number of +1's , 0's, and -1's satisfying the matching condition.

Equation (2) is used to find $X = X_{\text{thr}}$ value that maximizes the matching probability of an FM row. After finding X_{thr}, we can determine IR$_{\text{thr}}$ defined in Equation (3). As we previously state, we apply morphing to the CM rows having smaller IR values than IR$_{\text{thr}}$. We elucidate this with the following example.

Example 1: Consider a given logic function $f = x_1x_2x_6 + \overline{x_1}x_3\overline{x_4}x_5 + x_2\overline{x_5}x_6 + x_2\overline{x_3}\,\overline{x_5}x_6 + \overline{x_2}x_4x_5\overline{x_6}$ having 5 products and 12 literals (x's and their negations). Accordingly, the FM should have 5 rows and 12 columns, so every product is denoted with a row having $n = 12$ elements. Suppose that $p_o = 4\%$ and $p_c = 12\%$. As a result, Equation (1) results in

$$p_m(X) = \sum_{k=0}^{X} \sum_{t=0}^{12-X} \frac{12!}{k!\,(12-k-t)!\,t!} (0.12)^k (0.84)^{n-k-t} (0.04)^t$$

.

Using Equation (2), we calculate the $p_m(X)$ values as follows:

$$p_m(1) = 84\%,\ \ p_m(2) = 97\%,\ \ p_m(3) = 99.7\%$$
$$p_m(4) = 99.8\%,\ \ p_m(5) = 97\%,\ \ p_m(6) = 78\%,$$

and finally using Equation (3), we decide $X_{\text{thr}} = 4$ and IR$_{\text{thr}} = \frac{4}{12}$. It means that a row having a smaller IR value than $\frac{4}{12}$ needs morphing. Let's check the IR values of the rows denoting the products:

$P_1 = x_1x_2x_6$ has 3 literals, so the corresponding FM row has 3 +1's with IR $= \frac{3}{12} <$ IR$_{\text{thr}}$. Therefore, it needs to be morphed.

$P_2 = \overline{x_1}x_3\overline{x_4}x_5$ has 4 literals, so the corresponding FM row has 4 +1's with IR $= \frac{4}{12} =$ IR$_{\text{thr}}$. Therefore, morphing is not needed.

$P_3 = x_2\overline{x_5}x_6$ has 3 literals, so the corresponding FM row has 3 +1′s with IR= $\frac{3}{12} < \mathrm{IR_{thr}}$. Therefore, it needs to be morphed.

$P_4 = x_2\overline{x_3}\ \overline{x_5}x_6$ has 4 literals, so the corresponding FM row has 4 +1′s with IR $= \frac{4}{12} = \mathrm{IR_{thr}}$. Therefore, morphing is not needed.

$P_5 = \overline{x_2}x_4x_5\overline{x_6}$ has 4 literals, so the corresponding FM row has 4 +1′s with IR $= \frac{4}{12} = \mathrm{IR_{thr}}$. Therefore, morphing is not needed.

After determining which rows need to be morphed, we start with the row having the smallest IR and its corresponding product. In order to increase the IR value of the row, new redundant literals need to be added to the product such that the resulted new function should equal to the given function. Another restriction is that we can only add literals found in the given function to avoid adding extra wire/line resulting in a larger array.

First, we add maximum number of literals to reach $\mathrm{IR_{thr}}$ and then we check if the new function equals to the given function. If the equivalence is satisfied, the given function is updated with the new product; otherwise we remove a literal and check the equivalence again. This process is continued until all of the products to be morphed are considered. We elucidate this with the following example.

Example 2: Consider a same logic function used in the previous example: $f = x_1x_2x_6 + \overline{x_1}x_3\overline{x_4}x_5 + x_2\overline{x_5}x_6 + x_2\overline{x_3}\ \overline{x_5}x_6 + \overline{x_2}x_4x_5\overline{x_6}$. From the previous example, we know that P_1 and P_3 need to be morphed.

$P_1 = x_1x_2x_6$ has IR $= \frac{3}{12}$, so only one literal can be added because $\mathrm{IR_{thr}} = \frac{4}{12}$. Candidate literals are x_3, x_4, x_5, $\overline{x_3}$, $\overline{x_4}$, and $\overline{x_5}$. After checking all these options, we find that only P_1x_5 satisfies the equivalence. After updating the given function, we continue with P_3.

$P_3 = x_2\overline{x_5}x_6$ has IR $= \frac{3}{12}$, so only one literal can be added as well. Candidate literals are x_1, x_3, x_4, $\overline{x_1}$, $\overline{x_3}$,and $\overline{x_4}$. After checking all these options, we find that only P_3x_3 satisfies the equivalence. We update our the given function again and stop. As a result, the final function is $f = x_1x_2x_5x_6 + \overline{x_1}x_3\overline{x_4}x_5 + x_2x_3\overline{x_5}x_6 + x_2\overline{x_3}\ \overline{x_5}x_6 + \overline{x_2}x_4x_5\overline{x_6}$.

A pseudocode of our complete morphing algorithm is given in Algorithm 2.

Note that the existing morphing methods are utilized during the mapping process and check the equivalence of a given logic function and the crossbar realization of the same logic function. Therefore they should be performed in every step of the process; they are highly algorithm dependent. On the contrary, our approach performs the logic morphing process only one time and it is algorithm independent. Morphed or manipulated logic functions found with our method can be fed to any other algorithm as long as defect parameters satisfy required conditions.

Algorithm 2 Pre-mapping Morphing

```
Input: Row IRvalues of $\mathrm{FM_{m\times n}}$, logic function ($f$) and $\mathrm{IR_{thr}}$
Output: Morphed $\mathrm{FM_{m\times n}}$
morphedproducts =[]

for k = 1 to m do                                  ▷ m : number of products
    IR[k] ← IRvalues[k]                            ▷ IR value of kth row
    if IR[k] < $\mathrm{IR_{thr}}$ then
        morphedproducts[k] = k
    end if
end for

morphedproducts[k] ← morphedproducts[k] sorted with IR[k]'s in ascending order

for k = morphedproducts do
    candidateproducts = generate_products ($\mathrm{IR_{thr}}$)     ▷ products with $\mathrm{IR_{thr}}$
    for t = candidateproducts do
        $P_k$ = candidateproducts[t]
        $f^* = f$ with a candidateproduct ($P_k$)
        if $f^* = f$ then
            $f = f^*$
            break
        end if
    end for
end for
FM = generate_functionmatrix ($f$)                 ▷ FM of the new $f$
```

3.2 Function and Crossbar Matrix Sorting

For the matrices to be sorted, the number of columns is always less than or equal to the number of rows. In case, an FM or a CM does not satisfy this, it is transposed. The reason of this operation is to increase the matching probability of two rows. Indeed, for almost all benchmark circuits, transpose operation is not needed since the circuits generally have more products than literals.

The purpose of sorting is to decrease the chance of probable mismatches seen in the next step of the algorithm. We apply different sorting methods to the FM and the CM that is originated due to the different element types found in these matrices. The FM and the CM have two {+1, -1} and three {+1, 0, -1} distinct elements, respectively. Therefore, while sorting with one type of element is adequate for the FM, two elements should be considered for the CM.

First, we apply sorting to the rows of the matrices. For the FM, they are sorted according to the number of dominant defect type in descending order. Since it is harder to match dominant defects, the row having the most is assigned first. As for the CM, rows are sorted with respect to the number of defective elements in descending order as well. Thus, the most defective row or the row having the least number of functional switches is placed at the top of the CM, so the most problematic rows have a priority in the matching process.

Second, we apply sorting to the columns of the matrices. We sort the columns of the FM according to the number of dominant defects in descending order. Figure 5(a) shows the sorted final FM. Columns of the CM are sorted according to the dominant column indices in descending order. Recall that first index/2-tuple element represents the number dominant defects. Figure 5(b) shows our column sorting method. Note that although the third column of the given

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TETC.2017.2755458, IEEE Transactions on Emerging Topics in Computing

CM with an index of (0,2) has more defective elements than the sixth column with an index of (1,0), (0,2) comes after (1,0) after sorting is performed. The reason behind the dominant column index based sorting is to increase row matching probabilities. Note that we apply different sorting techniques for the rows and the columns of the CM since our matching method, applied in the next step of the algorithm, is a one dimensional row matching instead of a conventional two dimensional matrix matching.

Another important aspect of our sorting method is that, a dominant defect type is determined not only considering defect rates, but as well as an IR value of a given function as defined in Section 2. It means that a dominant defect type does not necessarily have an higher defect rate. The reason behind is that matching probabilities of matrices are directly affected by defect rates and IR values. Consider an FM with a given IR and a CM with given defect rates. Matching probability of stuck-closed defects is negatively correlated with the stuck-closed defect rate, and positively correlated with the IR. Similarly, matching probability for stuck-open defects has a negative and a positive correlation with the stuck-open defect rate and the (1-IR), respectively.

Finally, after assigning the sorted form of the columns of the FM to the IA, the algorithm proceeds to the next step.

3.3 Matching and Backtracking

The algorithm performs row by row matching between the sorted matrices advancing from top to bottom. During the process, matched rows of the CM are traced with an array showing which rows of the FM are assigned to them. At first, the matching searches only unmatched rows. If an FM row can not be matched with the unmatched rows of the CM, then backtracking starts by considering the matched rows of the CM from top to bottom. If a matching is found, the previously assigned row of the FM is checked once whether it can be assigned to an unmatched row of the CM. If this check results in a mismatch then the algorithm continues with the next matched row of the CM and repeats the same process to find a valid matching. This failsafe condition does prevent the algorithm to stop prematurely. This is also demonstrated in Section 4.

This final step of the algorithm is illustrated in Figure 6. Final assignments of the IA and the OA are also given in the figure.

3.4 Algorithm Evaluation

Consider a given function with m products and n literals resulting in an FM with a size of $m \times n$. The fist step of the algorithm finds equivalent products of a logic function with higher IR values. For every product, either a literal can be added or removed. Since there are n literals, total possibilities are 3^n in the worst-case scenario. This process is performed for every product which makes the number of operations $m.3^n$. As a result, we have an exponential time complexity $O(3^n)$. However, because this method is executed only once and is applied to a small portion of the products, overhead runtime cost is not dominant considering the other steps of the algorithm.

Following steps of the algorithm aim to find valid input and output assignments. Solution space of the problem as the total number of input and output assignments is $m!.n!$ which is computationally intractable for large matrices. However, as observed in [24] and [13] if an initial assignment is made for inputs with performing a one dimensional matching, for our case row by row matching, then the computational load drastically decreases.

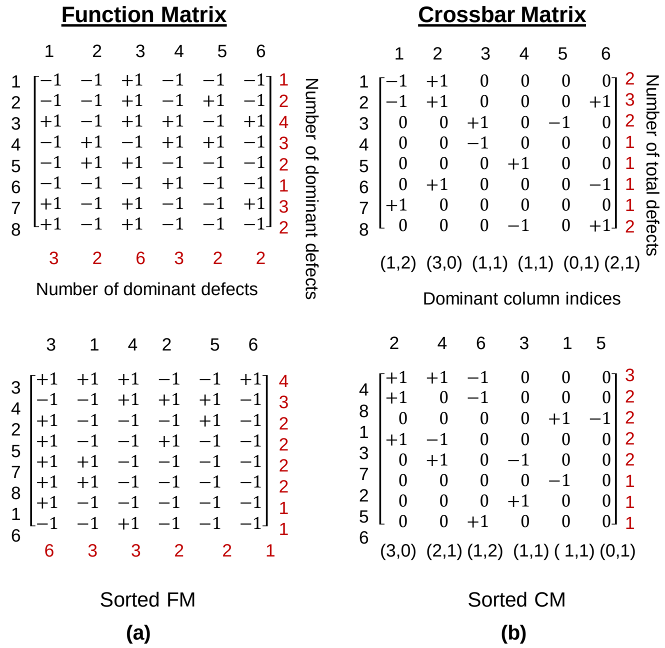
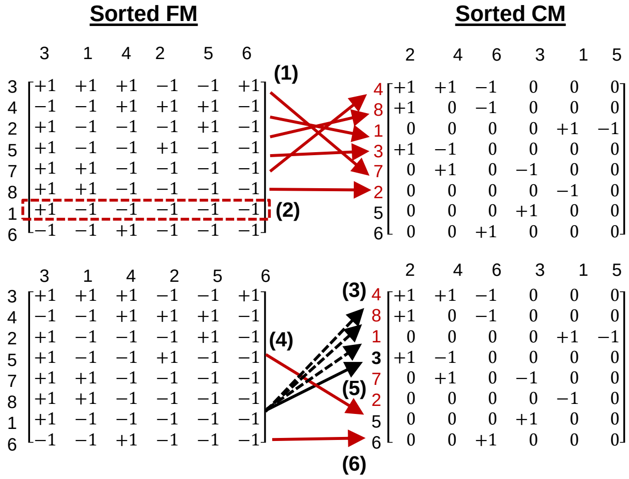


Fig. 5. (a) rows and columns of the FM are sorted according to the dominant defect type (b) rows and columns of the CM are sorted according to number of total defects and the dominant column index.



Steps:

(1) Row by row matching

(2) No matching found in unmatched rows for the 7th row

(3) Backtracking searches matched rows of the CM and unmatched rows for new assignment

(4) Previously assigned row of the FM is updated with an unmatched row

(5) The 7th row is assigned to the one of the matched rows

(6) Row by row matching is completed and **OA** is constructed

$$\textbf{IA} = [x_5\ x_3\ x_2\ x_1\ x_6\ x_4\ ]$$
$$\textbf{OA} = [P_4\ P_8\ P_1\ P_7\ P_5\ P_6\ P_3\ P_2\ ]$$

Fig. 6. Row by row matching and backtracking of the FM and the CM.

In order to generate an initial input assignment, we use

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TETC.2017.2755458, IEEE Transactions on Emerging Topics in Computing

TABLE 2
Success rate (%) and Runtime Values (in seconds) of the Simsir' algorithm [24], KNS-2DS [17], Yuan's algorithm [16] and the Proposed Algorithm (**PA**) for 1.5 Larger Crossbar Sizes with Different Defect Rates.

Circuit	**Size**	**IR**	$p_o = 12\%, p_c = 4\%, p_w = 4\%$								$p_o = 8\%, p_c = 8\%, p_w = 4\%$								$p_o = 4\%, p_c = 12\%, p_w = 4\%$							
			Psucc (%)				**Time (s)**				**Psucc (%)**				**Time (s)**				**Psucc (%)**				**Time (s)**			
			[24]	[17]	[16]	**PA**	[24]	[17]	[16]	**PA**	[24]	[17]	[16]	**PA**	[24]	[17]	[16]	**PA**	[24]	[17]	[16]	**PA**	[24]	[17]	[16]	**PA**
5xp1	75 x 14	28%	100	100	100	100	3.87	0.02	0.018	0.01	-	100	100	100	>60	0.03	0.021	0.01	-	34	100	100	>60	0.96	0.017	0.04
inc	34 x 14	39%	100	100	100	100	0.42	0.043	0.000	0.000	88	74	100	100	1.57	0.13	0.003	0.001	72	34	100	100	1.81	0.31	0.002	0.001
clip	167 x 18	29%	-	100	100	100	>60	0.12	0.155	0.01	-	94	100	100	>60	0.52	0.167	0.03	-	2	100	100	>60	5.84	0.15	0.091
misex1	32 x 16	24%	28	83	100	100	2.50	0.09	0.002	0.001	0	8	100	94	2.61	0.39	0.083	0.02	0	0	82	27	2.35	0.41	3.15	0.07
misex2	50 x 29	16%	0	0	100	86	11.36	0.63	0.266	0.03	0	0	0	0	8.38	0.66	9.93	0.06	0	0	0	0	7.52	0.68	8.81	0.093
sqrt8	40 x 16	25%	48	100	100	100	4.40	0.01	0.0041	0.009	0	42	100	100	5.00	0.33	0.007	0.02	0	0	99	90	4.42	0.59	0.75	0.54
9sym	87 x 18	33%	100	100	100	100	14.74	0.03	0.028	0.001	-	96	100	100	>60	0.11	0.027	0.002	-	26	100	100	>60	1.57	0.026	0.003
bw	65 x 10	35%	100	100	100	100	12.57	0.01	0.024	0.007	-	100	100	100	>60	0.021	0.023	0.008	-	100	100	100	>60	0.023	0.027	0.032
rd53	32 x 10	45%	100	100	100	100	0.01	0.008	0.002	0.001	100	100	100	100	0.14	0.006	0.002	0.000	100	100	100	100	0.25	0.006	0.002	0.001
rd73	141 x 14	42%	-	100	100	100	>60	0.07	0.093	0.002	-	100	100	100	>60	0.082	0.12	0.003	-	100	100	100	>60	0.07	0.1	0.006
sao2	58 x 20	38%	0	96	100	100	11.02	0.06	0.015	0.01	0	61	100	100	10.08	0.08	0.056	0.06	0	0	100	77	15.46	1.04	0.61	0.17
table5	158 x 34	36%	-	0	100	92	>60	6.83	0.485	0.01	-	0	0	0	>60	6.70	11.57	0.092	-	-	0	0	>60	6.89	11.21	0.58
t481	481 x 32	30%	-	77	96	100	>60	5.62	4.176	0.05	-	-	-	0	>60	5.78	>60	0.27	-	0	-	-	>60	5.92	>60	0.42

(> 60) means algorithm is not able to find a valid mapping under our runtime constraint of 60 seconds. For this reason we use - to denote an unknown success rate.

matrix sorting according to row and column elements. For a given $FM_{m\times n}$, the total number of elements need to be visited is $m.n$. After we construct the arrays showing the number of row and column elements, a quicksort of arrays yields $m.\log m$ and $n.\log n$ operations which makes the worst-case operation cost of this assignment process as $m.n + m.\log m + n.\log n$ resulting in a time complexity of $O(m.n)$.

After the initial assignment is determined, we match matrices row by row with backtracking. Every row has n elements, so Hadamard multiplication and checking for mismatches performs $n + n = 2n$ operations. Additionally, each function row is matched with m crossbar rows, so $2n.m$ operations are needed. In case of backtracking with checking matched rows, $m.m$ rows also need to be checked that results in $2n.(m + m^2)$ operations. For all of the matrix rows there are $m.2n.(m + m^2)$ operations in the worst-case scenario, so a time complexity becomes $O(n.m^3)$.

4 EXPERIMENTAL RESULTS

In this section, we give experimental results of the proposed algorithm (**PA**) in comparison with the Simsir's algorithm using partial graph construction and maximum bipartite mathcing [24], k-neighbour sort with 2-dimensional sort algorithm (**KNS-2DS**) [17], Yuan's algorithm using a memetic fitness approximation during matching process [16], and an **exact** algorithm using the Hungarian method in [14]. To our knowledge, KNS-2DS is the only algorithm that tailors itself according to the effects of multiple defect types. However, we also include the rest of the existing methods in compatible with multiple-type defects.

We prefer the Hungarian method as an exact algorithm due to its high efficiency as well as its similarities with the PA regarding the used matrix based models which yields a fair comparison. We also evaluate our morphing technique in comparison with the hardening and morphing techniques proposed in [18].

We use MATLAB$^{\text{TM}}$ to implement all of these algorithms and techniques with calling the Berkeley tool ABC [26]. We use standard benchmark circuits presented in [27]. All experiments run on a 3.4GHz Intel Core i7 CPU (only single core used) with 8GB 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/fb/TETC_Multiple_Type_Defect_Tolerance_Codes.rar

4.1 The PA versus the Existing Methods

We use three different defect types and generate 200 random crossbar matrices for each mapping to prevent runtime and success rate (**Psucc**) fluctuations. In addition, we limit the runtime as 60 seconds for each sample of 200 cases. We use 1.5 larger crossbar sizes than the optimal sizes that is a common practice in the literature [17], [18], [21], and [16]. We choose wire breakdown and bridging defect rates as 2%, so the total wire defect rate is $P_w = 4\%$.
Defect rates used in our simulations are listed as follows.

- Stuck-open $p_o = 12\%$, stuck-closed $p_c = 4\%$, and $p_w = 4\%$.
- stuck-open $p_o = 8\%$, stuck-closed $p_c = 8\%$, and $p_w = 4\%$.
- Stuck-open $p_o = 4\%$, stuck-closed $p_c = 12\%$, and $p_w = 4\%$.

The reason behind using different defect rates is to show the adapting performance of the algorithms for different defect orientations. In Table 2, comparison of the all algorithms and the PA is given. Considering the runtime values, the PA always yields best results. The slowest one is Simsir's algorithm due to its constant call of bipartite maximum matching routine. As for the Psucc values, the PA is the best one; Yuan's algorithm is better in certain cases at the cost of high runtime values. In short, tradeoff between runtime and success rate is directly related to the number of column permutations tried for mapping. The

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TETC.2017.2755458, IEEE Transactions on Emerging Topics in Computing

TABLE 3
Succes Rate and Runtime Comparison of the PA and the Exact Algorithm.

			$p_o = 12\%, p_c = 4\%, p_w = 4\%$			
			PA		Exact Algorithm	
Circuit	Size	IR	Psucc	Time	Psucc	Time
5xp1	75 x 14	28%	100%	0.01	100%	0.16
inc	34 x 14	39%	100%	0.00	100%	0.012
clip	167 x 18	29%	100%	0.01	100%	1.67
misex1	32 x 16	24%	100%	0.001	100%	0.017
misex2	50 x 29	16%	86%	0.03	100%	0.021
sqrt8	40 x 16	25%	100%	0.009	100%	0.025
9sym	87 x 18	33%	100%	0.001	100%	0.03
bw	65 x 10	35%	100%	0.007	100%	0.108
rd53	32 x 10	45%	100%	0.001	100%	0.009
rd73	141 x 14	42%	100%	0.002	100%	1.24
sao2	58 x 19	38%	100%	0.01	100%	0.053
table5	158 x 34	36%	92%	0.01	98%	0.52
t481	481 x 32	30%	100%	0.05	100%	53.08

PA finds a single column permutation (initial assignment) using sorting methods and the Yuan's algorithm tests the performance of many column permutations according to an objective function. For this reason, while runtime increases, success rate improves as well. Similarly, if multiple column permutations were used in the PA, its success rate could be improved. Another point is that for benchmarks *misex2*, *table5*, and *t481*, all algorithms fail when $p_o \leq p_c$. The common thing for these benchmarks is their low IR values that severely complicates stuck-closed defect tolerance.

4.2 The PA versus the Exact Algorithm

To find a valid mapping, we make an initial assignment to inputs of a crossbar and progressively construct an output assignment with row by row matching and backtracking. In this section, we evaluate our approach of finding the output assignment in comparison with an exact algorithm that constructs a matching matrix, previously defined in Section 2, and uses the Hungarian method to minimize the assignment cost which needs to be zero for a valid mapping.

Table 3 gives the results. The PA is able to maintain the same success rate as the exact algorithm for almost all cases. Only, for the benchmark *misex2* and *table 5*, there is a 14% and 8% success rate difference respectively. In terms of runtime values, our algorithm overwhelms the exact algorithm as expectedly. Additionally, Figure 7 gives comparisons of the algorithms in terms of success rate and runtime. For small logic functions, runtime difference is between 10 and 100 times. However as the size of a given function increases, runtime of the exact algorithm rapidly reaches very high values.

Note that both the PA and the exact algorithm operate with only one initial assignment or column permutation as opposed to the Yuan's algorithm that uses many different permutations. Therefore, here our comparison is fair for evaluation of the algorithms' matching performances.

4.3 Evaluation of the Morphing

We first justify that why our algorithm does not use logic hardening, but only use logic morphing. A hardening method improves tolerance of stuck-open defects by adding redundant literal lines that results in larger crossbar sizes [18].

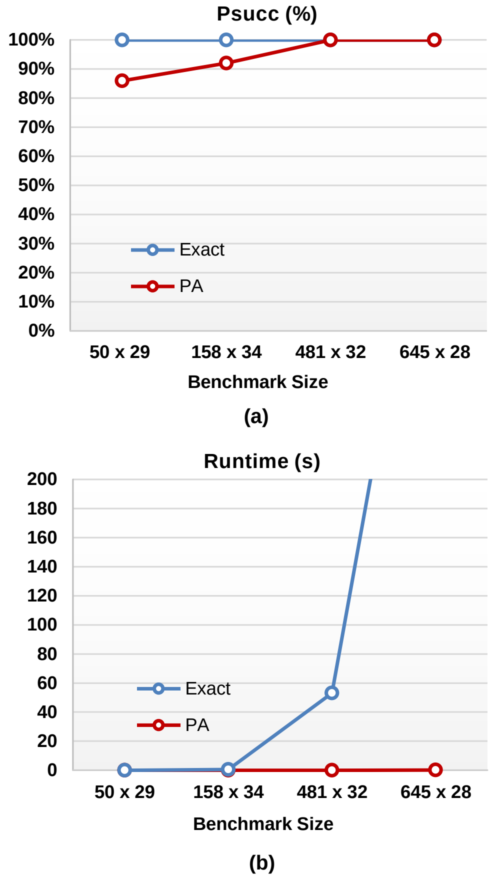


Fig. 7. Comparison of the PA and the exact algorithm showing (a) successful mappings found (b) runtimes when $P_o = 12\%, P_c = 4\%$, and $P_w = 4\%$.

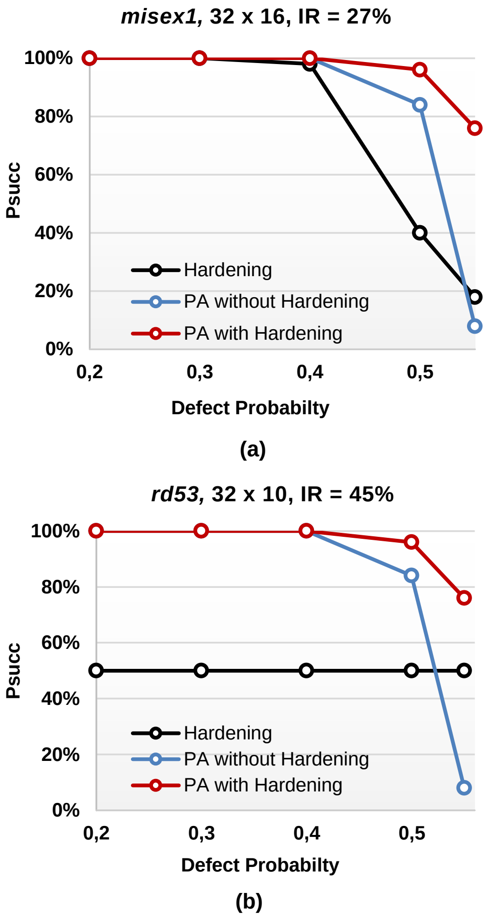
In Figure 8, we give results for the benchmarks *misex1* and *rd53* since they are thoroughly examined in [18]. We compare the hardening technique proposed in [18] and our PA with and without applying hardening. To effectively show the benefits of hardening, we select much higher rates for stuck-open defects than those for stuck-closed defects; stuck-open and the stuck-closed rates are respectively 90% and 10% of the defect probability. For example, 0.5 defect probability yields $p_o = 45\%$ and $p_c = 5\%$. Crossbars used in all mapping trials have 2 and 3 times larger row and column sizes, respectively as done in [18]. With an advantage of using these large sizes, the PA is able to find a valid mapping without a need of hardening for defect probabilities under 50%. This is shown in Figure 8. Benefits of the hardening method is only apparent in case defect rates are higher than 50%. Nonetheless, it is reasonable to assume that defect rates are unlikely to be that high.

We also evaluate our pre-mapping morphing technique used in the PA in comparison with the PA without morphing and the tailored PA performing morphing during mapping. We select the benchmarks *misex1* and *sqrt8* having low IR values in order to see the effectiveness of morphing. Table 4 shows the results. Examining the numbers, we see that

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TETC.2017.2755458, IEEE Transactions on Emerging Topics in Computing

TABLE 4
Success rate (%) and Runtime Results of *Misex1* and *Sqrt8* using the Tailored PA Performing Morphing During Mapping, the PA without Morphing, and the PA using a Pre-mapping Morphing

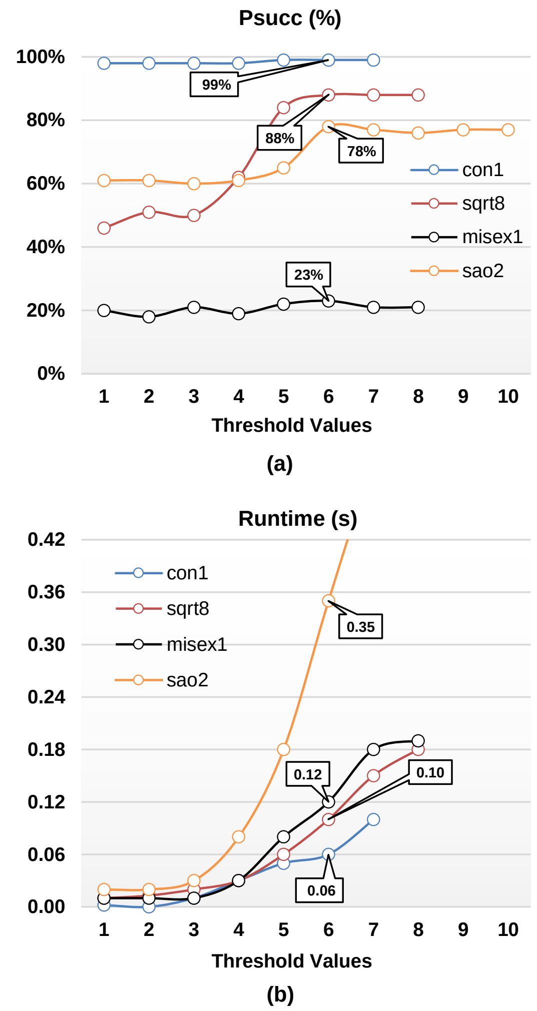
p_o	p_c	**Morphing during Mapping**				**PA without Morphing**				**PA**			
		misex1		*sqrt8*		*misex1*		*sqrt8*		*misex1*		*sqrt8*	
		Psucc	**Time**	**Psucc**	**Time**	**Psucc**	**Time**	**Psucc**	**Time**	**Psucc**	**Time**	**Psucc**	**Time**
10%	4%	100%	0.11	100%	0.37	100%	0.001	100%	0.001	100%	0.001	100%	0.01
10%	6%	98%	0.16	100%	0.53	100%	0.001	100%	0.001	100%	0.008	100%	0.015
10%	8%	88%	0.25	94%	0.83	98%	0.001	98%	0.001	96%	0.023	100%	0.027
10%	10%	52%	0.30	76%	1.04	59%	0.001	86%	0.007	74%	0.023	95%	0.028
10%	12%	4%	0.33	30%	1.45	14%	0.007	44%	0.008	22%	0.06	57%	0.035



(a)

(b)

Fig. 8. Success rates of the PA with and without hardening, and the hardening method in [18] for (a) *misex1* (b) *rd53* benchmark circuits.



(a)

(b)

Fig. 9. Success and runtime values of the PA with different threshold values, given in (a) and (b), respectively. Our threshold values (X_{thr}'s) found with the proposed equations are given as bubbles on the line.

our pre-mapping morphing technique increases the success rates up to 30%. Additionally, runtime overhead of our pre-mapping morphing is better than that of the during-mapping morphing. While our pre-mapping morphing is a one-time operation that can be used for different crossbars, applying morphing during mapping needs to be run for every different crossbar.

In addition, we evaluate our threshold values used during the morphing process. We choose 4 benchmark functions with low IR values such as *con1*, *sqrt8*, *misex1*, and *sao2*, and execute the algorithm with different threshold values. It is clear from Figure 9 (a) success rate of the algorithm stabilize after our threshold, however choosing a larger threshold increases the runtime drastically as shown Figure 9 (b). This approves our technique aiming to find an optimum threshold values (X_{thr}) for maximum matching probability.

4.4 Unsuccessful Cases and Underlying Cause

Experimental results indicate that for certain benchmark circuits with high stuck-closed defects rates, the PA is not able to find a valid assignment. In order to determine whether this phenomena is particular to the PA or originated from the difficulty of mapping for these specific cases, we utilize the exact algorithm. Although our iterations reach as high as 10^8 different input assignments, the exact algorithm is not able to produce a single assignment with a zero cost or a valid mapping.

When we analyze matching matrices of trials, we realize that some function rows have no valid matching with any of the rows in the crossbar. Although input assignment iterations are increased to construct sparser matching matrices meaning more rows can be matched with each other, it is also not possible to find a valid mapping with the exact method. We believe that this is the main reason why the PA and the rest of the algorithms yield 0% success rate for benchmark circuits such as *misex2*, *table5*, and *t481*.

5 CONCLUSION

In this paper, we present a fast heuristic algorithm using pre-mapping morphing, defect oriented sorting, row by row matching, and backtracking for defect tolerant logic mapping of nano-crossbars. We show that our algorithm covers both crosspoint defects including stuck-open and stuck-closed types and wire defects including bridging and broken types. Effectiveness of the algorithm is demonstrated through extensive simulation results covering a wide range of comparison with existing methods using industry standard benchmarks. Especially, it is shown that our algorithm independent pre-mapping morphing considerably improves the tolerance of stuck-closed defects.

In future work, we plan to extend our studies to cover both variation and defect characteristics of nano-crossbar arrays. Another future direction is developing the proposed techniques for memristive and resistive crossbars which are very similar to our current architecture interest.

REFERENCES

[1] W. Lu and C. M. Lieber, “Nanoelectronics from the bottom up,” *Nature materials*, vol. 6, no. 11, pp. 841–850, 2007.

[2] G. Bourianoff, J. E. Brewer, R. Cavin, J. A. Hutchby, and V. Zhirnov, “Boolean logic and alternative information-processing devices,” *Computer*, vol. 41, no. 5, pp. 38–46, 2008.

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

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

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

[8] 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.

[9] 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.

[10] 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.

[11] M. M. Ziegler and M. R. Stan, “Design and analysis of crossbar circuits for molecular nanoelectronics,” in *Nanotechnology, 2002. IEEE-NANO 2002. Proceedings of the 2002 2nd IEEE Conference on*. IEEE, 2002, pp. 323–327.

[12] M. B. Tahoori, “Application-independent defect-tolerant crossbar nano-architectures,” in *Proceedings of the 2006 IEEE/ACM international conference on Computer-aided design*. ACM, 2006, pp. 730–734.

[13] 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.

[14] H. W. Kuhn, “The hungarian method for the assignment problem,” *Naval research logistics quarterly*, vol. 2, no. 1-2, pp. 83–97, 1955.

[15] 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.

[16] 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.

[17] 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.

[18] 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.

[19] Y. Zheng and C. Huang, “Defect-aware logic mapping for nanowire-based programmable logic arrays via satisfiability,” in *Proceedings of the Conference on Design, Automation and Test in Europe*. European Design and Automation Association, 2009, pp. 1279–1283.

[20] 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.

[21] 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.

[22] 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.

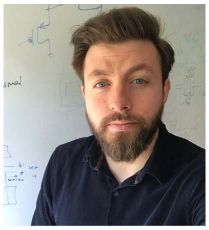
[23] H. Naeimi and A. DeHon, “A greedy algorithm for tolerating defective crosspoints in nanopla design,” in *Field-Programmable Technology, 2004. Proceedings. 2004 IEEE International Conference on*. IEEE, 2004, pp. 49–56.

[24] 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.

[25] O. Tunali and M. Altun, “Permanent and transient fault tolerance for reconfigurable nano-crossbar arrays,” *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 36, no. 5, pp. 747–760, 2017.

[26] B. L. Ssynthesis, “Verification group,” *ABC: A system for sequential synthesis and verification*, 2013.

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



Onur Tunalı received his BSc degree in mathematics at Istanbul University and MSc degree in Nanoscience and Nanoengineering at Istanbul Technical University. He has worked as a researcher in various projects including TUBITAK and EU H2020 RISE projects. His current research interests include logic synthesis, reconfigurable nano-crossbars, algorithm design, emerging computing, and reliability.

Mustafa Altun received his BSc and MSc degrees in electronics engineering at Istanbul Technical University in 2004 and 2007, respectively. He received his PhD degree in electrical engineering with a PhD minor in mathematics at the University of Minnesota in 2012. Since 2013, he has served as an assistant professor at Istanbul Technical University and runs the Emerging Circuits and Computation (ECC) Group. Dr. Altun has been served as a principal investigator/researcher of various projects including EU H2020 RISE, National Science Foundation of USA (NSF) and TUBITAK projects. He is an author of more than 30 peer reviewed papers and a book chapter, and the recipient of the TUBITAK Success, TUBITAK Career, and Werner von Siemens Excellence awards.