A Fast Hill Climbing Algorithm for Defect and Variation Tolerant Logic Mapping of Nano-Crossbar Arrays

Furkan Peker and Mustafa Altun

Abstract—Nano-crossbar arrays are area and power efficient structures, generally realized with self-assembly based bottom-up fabrication methods as opposed to relatively costly traditional top-down lithography techniques. This advantage comes with a price: very high process variations. In this work, we focus on the worst-case delay optimization problem in the presence of high process variations. As a variation tolerant logic mapping scheme, a fast hill climbing algorithm is proposed; it offers similar or better delay improvements with much smaller runtimes compared to the methods in the literature. Our algorithm first performs a reducing operation for the crossbar motivated by the fact that the whole crossbar is not necessarily needed for the problem. This significantly decreases the computational load up to 72% percent for benchmark functions. Next, initial column mapping is applied. After the first two steps that can be considered as preparatory, the algorithm proceeds to the last step of hill climbing row search with column reordering where optimization for variation tolerance is performed. As an extension to this work, we directly apply our hill climbing algorithm on defective arrays to perform both defect and variation tolerance. Again, simulation results approve the speed of our algorithm, up to 600 times higher compared to the related algorithms in the literature without sacrificing defect and variation tolerance performance.

Index Terms—Nano-crossbar Arrays; Variation Tolerance; Defect Tolerance; Worst-case Delay Optimization.

1 INTRODUCTION

Nano-crossbar arrays emerged as a new computing scheme with an aim of solving the longstanding miniaturization problems of CMOS circuits [1], [2], and [3]. Each crosspoint of an array is used as a switching element showing field-effect transistor (FET) like behaviour with programmability features [4], [5], and [6]. Therefore, nano-crossbar arrays operate similarly to conventional programmable logic arrays (PLA's) [7], [8], [9], [10], [11], and [12]. Structures of a conventional PLA and a nano-crossbar array are given in Figure 1. Any logic function f can be implemented with properly placed devices on AND/OR planes along with the corresponding input literals, for both conventional PLA's and nano-crossbars. For a given nano-crossbar array structure in Figure 1, each crosspoint is either a transistor working as a switch between the power supply and the ground or a short circuit. If there is a transistor on a crosspoint, the corresponding literal line controls this transistor from its gate to switch between ON and OFF states. In a case where any of the vertical lines has a connection to ground level, the output function f becomes logic 0.

Nano-crossbar arrays offer unique features such as small area, low power consumption, and easy manufacturability. Two fully operational crossbar implementations as a nanoprocessor and a finite-state machine are shown to be feasible in [13] and [14]. However, high process variations and defects are big headache that significantly effect circuit performances, especially for delays. Consider a function $f = x_1x_2 + x_2x_3 + x_1x_3$ to be implemented on a 3×3 nano-crossbar array having 3 output lines for 3 products and 3 input lines for 3 literals. Suppose that delay of each crosspoint varies between d and $10d$ where d is a minimum delay time. Here, 6 of 9 crosspoints should be configured as FET's, 2 on each product line, and the rest of them are configured as disconnected lines. Selection of these 6 crosspoints plays an important role for the worst-case delay. There are total of $3! \times 3! = 36$ options to select as the number of orderings of input and output lines, and each

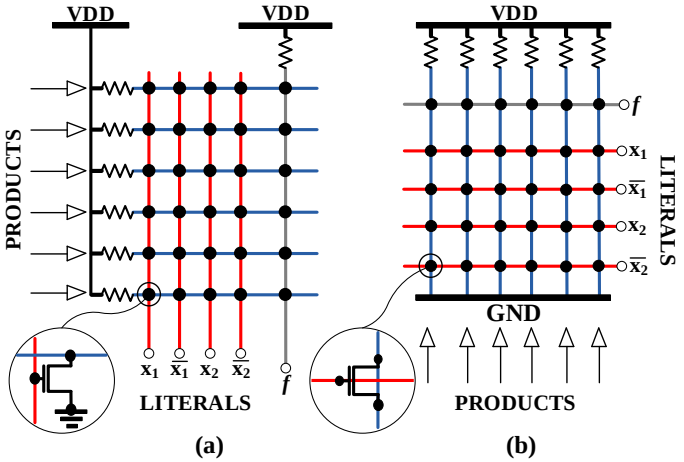


Fig. 1. Standard structures of (a) one-output programmable logic array (PLA), and (b) one-output nanocrossbar array. Each crosspoint is a switch operating as an n-type FET.

- *This work is part of a project that has received funding from the European Union's H2020 research and innovation programme under the Marie Skodowska-Curie grant agreement No 691178. This work is supported by the TUBITAK-Career project #113E760*
- *Furkan Peker and Mustafa Altun are with the Department of Electronics and Communication Engineering, Istanbul Technical University, Istanbul, Turkey, 34469.*
- *E-mails: {pekerf, altunmus}@itu.edu.tr*

selection gives a different delay value between $2d$ or $20d$, since there are 2 crosspoints on each product line. If we have a chance to measure delay contribution of each crosspoint, then by trying these 36 options we can find the best solution. However, with an increase in the number of input lines (n) and output lines (m), the number of options $n! \times m!$ quickly grows beyond practical limits. After the size of only 8×8, it is not feasible to use an exhaustive search. Indeed this problem, commonly known as variation tolerant logic mapping (VTLM) problem, is an NP-Complete problem [15].

In this work, we focus on the worst-case delay optimization problem in the presence of high process variations. We propose a fast hill climbing algorithm for the VTLM problem that improves the worst-case delay up to 25% with an average of 20% for our benchmark set and random generated matrices. We comment that the whole crossbar is not necessarily needed to be used for the worst-case delay optimization problem, so our algorithm first performs a reducing operation for the crossbar. This significantly decreases the computational load of the algorithm, up to 72% for our standard benchmark set. Next, initial column mapping is applied. After the first two steps that can be considered as preparatory, the algorithm proceeds to the last step of hill climbing row search with column reordering where optimization for variation tolerance is performed.

Since our algorithm primarily eliminates crosspoints with highest delay values, it can be also used for defect tolerance by assigning relatively high delay values on defective crosspoints. Thus, both defect and variation tolerance could be achieved. However, if defects are considered, the proposed matrix reducing approach, as the first step of the algorithm, can not be used since all defects spreading to the whole crossbar should be tolerated. By running our algorithm for defect and variation tolerance, we see that it gives up to 600 times lower runtimes compared to the related algorithms in the literature without sacrificing defect and variation tolerance performance. This certainly approves the efficiency of the proposed algorithm.

1.1 Previous Works

Although defect/fault tolerant logic mapping for nanocrossbars has been long studied [16], research on variation tolerance is relatively new. First, Gojman and Dehon consider variations on crosspoint transistor parameters to accurately determine the placement of defects as opposed to using randomly assigned defect maps [17]. They propose a post fabrication mapping algorithm (VMATCH) to tolerate defects caused by variations on threshold voltage values of crosspoint FET's. By using independent Gaussian distributions, they determine defects such that when an ON resistance of a crosspoint FET is larger than the OFF resistance, the corresponding crosspoint is defective. As a result, this work can be considered as a transition between defect and variation tolerance methods. However, it does not directly focus on variation tolerant performance optimization.

As a complete variation tolerance methodology, Tunc and Tahoori propose a logic mapping algorithm based on simulated annealing [18]. Additionally, they offer a delay testing technique on nanocrossbar arrays to obtain delay contributions of crosspoints that is needed for constructing a variation matrix. Since the algorithm uses randomly selected iterations without progress monitoring, its efficiency is questionable; sufficient results can not be achieved unless relatively high number of trials are reached. On the other hand, our algorithm is designed to make a continuous progress; that is why we call it a *hill climbing algorithm*. Another approach based on linear integer programming is proposed by Zamani et al. [15]. Although satisfactory delay results can be achieved by this systematic method, runtime values are even worse than those of the simulated annealing algorithm.

Yang et al. propose a different approach for the VTLM problem using a non-dominated sorting genetic algorithm [19]. While finding near Pareto optimal solutions, time overhead of this algorithm is disadvantageous. In average, runtimes are generally much higher ($\times 30 - 40$) than ours. In order to improve runtimes, Zhong et al. use a greedy reassignment technique [20], originally proposed in [21]. We also used this method as as an alternative to initial column mapping method and compared optimization results.

Another evolutionary algorithm is proposed by Zhong et al. [22]; it is a bi-level multi-objective optimization algorithm that uses different approaches on row and column mappings defined as lower and upper level problems. Every individual of an upper level problem is required to be first solved as a lower level problem that puts too much burden on the lower level (row order) algorithm. Therefore, the overall algorithm performance mostly depends on the performance of the lower level algorithm, defined as a min–max-weight and min-weight-gap bipartite matching problem being solved by a heuristic variant of the Hungarian method. A follow-up work, fundamentally based on the same approach, is also proposed [23]. It achieves both defect and variation tolerance using a memetic algorithm. For both of these algorithms, much better delay values are obtained compared to the algorithm in [19]. However, runtime is still an issue. Our algorithm gives delay values in the same range while having considerably lower runtimes.

There are also studies focusing on adding extra rows or columns for better variation tolerance at the cost of area yield. In their work, Zamani and Tahoori use row redundancies with duplicated input lines [24]. Their approach successfully reduces the critical path delay with an average of 50%. Along with the area overhead problem, these studies have a logical flaw: before fabrication, the amount of redundancies should be known, but it can be determined after fabrication (after post fabrication delay test).

Another factor of evaluation is the algorithms' capability to tolerate both defects and variances. Among the mentioned studies, the ones [18], [19], [22], and [23] can be either directly or with slight modifications applied for defect and variation tolerance. We consider all of these algorithms in the experimental results part. There are other algorithms targeting both defects and variances [25] and [26], but they are outperformed by the considered algorithms. Note that our algorithm is also directly applicable for defect and variation tolerance.

1.2 Organization

Organization of this work can be summarized as follows.

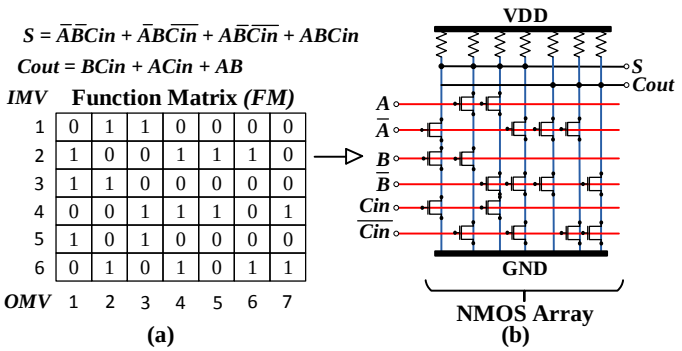


Fig. 2. Mapping steps: (a) target logic functions and their function matrix, and (b) logic mapping on a FET type nanoarray.

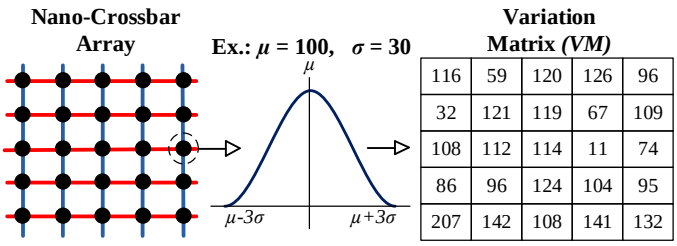


Fig. 3. Variance matrix generation for a 5×5 crossbar array using Gaussian distribution.

- We introduce preliminaries for the VTLM problem and define our performance objectives in Section 2.
- We propose our VTLM algorithm in Section 3 with its steps given in the subsections. In these steps, we use a *function matrix reducing*, an *initial column mapping*, and a *hill climbing* row search with *column reordering* methods. In this section, we also explain how to use the proposed algorithm for both defect and variation tolerance.
- Experimental results are given Section 4, and Section 5 concludes this study with insights and future directions.

2 PRELIMINARIES

In variation tolerant logic mapping scheme, a target logic function and a nano-crossbar are generally represented by matrices, called a function matrix FM and a variation matrix VM, respectively. The goal of any mapping algorithm is achieving a desired performance by determining the proper row and column orders for FM to be mapped on VM.

A binary matrix *FM* indicates a logic function to be mapped onto a nanoarray. As a general design topology, matrix rows and columns represent function literals and products, respectively. If a literal is included in a function product, intersection of the corresponding row and column is tagged as '1' (the crosspoint behaves as a switch); otherwise '0' (the crosspoint is an open circuit; crossed lines are disconnected). An example of FM and mapping scheme on a nano-crossbar array is given in Figure 2.

$$FM(i,j) = \begin{cases} 1, & \text{if } i\text{th literal is included in } j\text{th product} \\ 0, & \text{otherwise} \end{cases}$$

FM Parameters: number of rows m, number of columns n, input mapping vector IMV, output mapping vector OMV, and crosspoint ratio CR as the ratio of the number of 1's to the matrix size $m \times n$.

As a representation of a crossbar, *VM* has switching delay values of all crosspoints. These values are determined by delay testing methods [18], explained as follows .

Delay Testing: all crosspoints act as FET type switches and stay constant in non-controlling values (logic 1 for NAND and logic 0 for NOR type). In this state, for each input, tester applies falling and rising control signals while the other inputs stay at the same non-controlling value. Here, while switching an input i, transition delay value at the output j is observed as the delay value of the crosspoint located at (i, j). To generate VM for the VTLM process, average rising and falling transition times are used [18]. Note that if any crosspoint delay value is relatively high ($\times 10$) than other crosspoint delays or it does not switch at all, it is considered as a defective crosspoint, so defect tolerance methods applied.

For simulations, it is widely assumed that the measured delay values show a Gaussian (Normal) distribution with a mean μ, a standard deviation σ, and a coefficient of variation or relative standard deviation $COV = \sigma/\mu$ [18], [22], and [23]. An example of *VM* and a random delay value generation scheme are given in Figure 3.

$$VM(i,j) = \Big\{ \text{crosspoint delay } (j,i) \text{ with Gaussian}(\mu, \sigma^2)$$

VM Parameters: number of horizontal lines/wires m, number of vertical lines n. Note that the sizes of *FM* and *VM* are same; no extra crossbar redundancy is used that is a common practice in defect tolerance.

Performance Matrix (PM) is a Hadamart product of FM and VM matrices.

$$(PM)_{j,i} = (FM)_{j,i}(VM)_{j,i} \quad (1)$$

$$i = 1, 2, ...n, \quad j = 1, 2, ...m$$

Since the highest column delay, sum of all delay values in a column, represent the worst-case delay for a FET type nanoarray, we define a row matrix *FPM* as a FET performance matrix.

$$FPM_i = \Sigma_j (PM)_{j,i} \quad (2)$$

2.1 Objectives

In the literature, three main objectives are considered while developing variation tolerant delay optimization algorithms. The first one is minimizing the worst-case delay, so minimizing the highest valued column in FPM. The second one is maximizing the best-case delay, so maximizing the lowest valued column in FPM. Finally, the third one is minimizing the difference between the worst-case and best-case delays. Definitions of these objectives are given in Equations 3 , 4, and 5.

$$Objective\ 1 = minimize(maximum(FPM_i)) \quad (3)$$

$$Objective\ 2 = maximize(minimum(FPM_i)) \quad (4)$$

$$Objective\ 3 = minimize(Objective\ 1\ -\ Objctive\ 2) \quad (5)$$

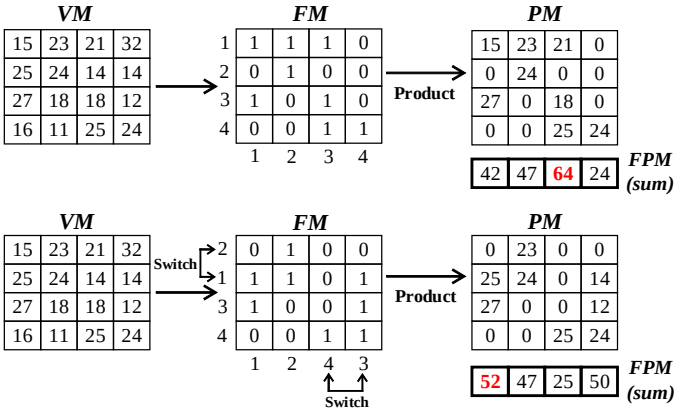


Fig. 4. An example of logic mapping optimization by interchanging row and column pairs.

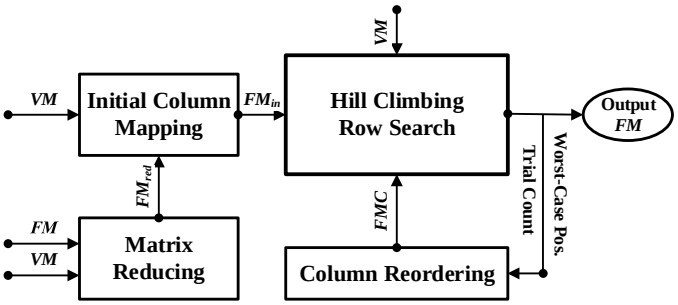


Fig. 5. Work flow of the proposed hill climbing algorithm.

In our work, we focus on $Objective$ 1 since the worst-case delay optimization is by far the mostly desired and used one in the literature. On the other hand, $Objective$ 2 is generally used for defective crossbars to especially take into account stuck-at zero defects. Additionally, $Objective$ 3 is valid for very specific applications requiring an almost constant delay values.

Our general logic mapping scheme for worst-case delay optimization is given in Figure 4. Here, we use interchangeability of rows and columns of the function matrix and find a better mapping to achieve $Objective$ 1.

3 PROPOSED ALGORITHM

In this section we introduce all stages of our hill climbing algorithm including function matrix reducing, initial column mapping, and hill climbing row search with column reordering. As a pre-processing method, we first use our FM reducing method that finds unnecessary columns to be excluded.

Next we start the mapping process. Here, the only tool that we have is interchanging rows and columns of FM. Since total delay values in a column determines the worst-case delay, changing rows or columns have different effects on $Objective$ 1. Therefore, we should determine whether dealing with rows or columns first. It is obvious that row ordering followed by column ordering is constructive; performance improvement is guaranteed. On the other hand, performing column ordering after row ordering is destructive; ordering columns kills our initial effort of ordering rows. As a result, we first perform column mapping and then precise tuning by row search. We repeat this process

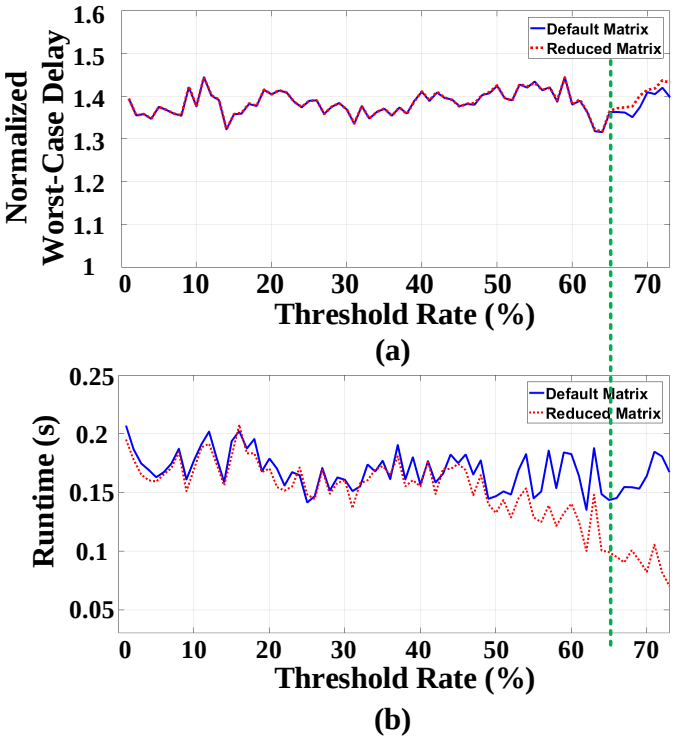


Fig. 6. Results using reduced and standard FM's for (a) worst-case delays, and (b) runtimes for 32×32 random matrix with 40% CR.

using column reorderings until a maximum number of reorderings is reached.

Work flow of the proposed algorithm scheme is given in Figure 5. The steps of the algorithm are given in the following three subsections. Fourth subsection analyses probabilistic context of our hill climbing algorithm. The fifth subsection explains how to use the proposed algorithm for both defect and variation tolerance.

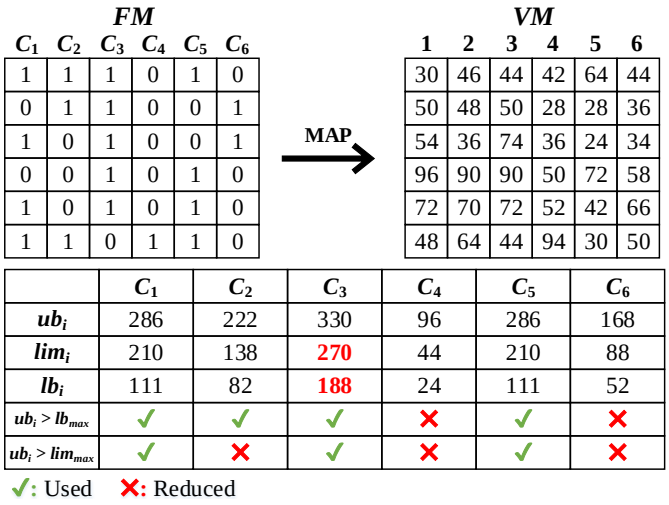
3.1 Function Matrix Reducing

Since we are only interested in the highest valued column in FPM, relatively low valued columns having low transistor counts which do not determine the worst-case delay performance, are not needed. An example is given in Figure 6. Here, we use a threshold as a percentage of the maximum number of 1's in a column of FM. We remove columns having values under this threshold from FM. Then we perform our algorithm with using the reduced form of FM to achieve $Objective$ 1, and compare the results with those obtained with a standard FM, not reduced. The figure tells us that until the threshold of nearly 65%, there is no difference between the worst-case delay values. On the other hand, algorithm runtime for the reduced algorithm decreases as the threshold increases since matrix is getting smaller. As a result, by matrix reducing, we can achieve 35% runtime improvement without any degradation for the worst-case delay for this specific example. Motivated by this, we propose a method that effectively finds the threshold.

First, for each column of FM, we determine lower and upper bounds on delay values as well as a specific limit value. Suppose that the ith column C_i of FM has a transistor count of T_i, representing the number of 1's.

Algorithm 1 Proposed Matrix Reducing (Using lim_{max} as Bound).

Input: $FM_{m\times n}$, $VM_{m\times n}$
Output: $FM_{reduced}$
$FM_{reduced} \leftarrow FM$
for each FM column i **do**
  $T_i \leftarrow$ transistor count of C_i
  for each VM column j **do**
    $All\_ub_{i,j} \leftarrow$ sum of the highest T_i values on VM_j
    $All\_lb_{i,j} \leftarrow$ sum of the lowest T_i values on VM_j
  end for
  $ub_i \leftarrow$ maximum of All_ub_i
  $lim_i \leftarrow$ maximum of All_lb_i
end for
$lim_{max} \leftarrow$ maximum of lim_i
for each FM column i **do**
  if $ub_i < lim_{max}$ **then**
    reduce C_i from $FM_{reduced}$
  end if
end for



	C_1	C_2	C_3	C_4	C_5	C_6
ub_i	286	222	330	96	286	168
lim_i	210	138	270	44	210	88
lb_i	111	82	188	24	111	52
$ub_i > lb_{max}$	✓	✓	✓	✗	✓	✗
$ub_i > lim_{max}$	✓	✗	✓	✗	✓	✗

✓: Used ✗: Reduced

Fig. 7. An example for determining columns to be reduced considering lb_{max} and lim_{max} parameters.

For each column of VM, minimum and maximum sum of T_i elements are determined. Considering that VM has n columns, we have n minimum sum and n maximum sum values for each FM column. The lower bound lb_i for the ith column C_i is the lowest value of the n minimum sum values of C_i. The upper bound ub_i is the highest value of the n maximum sum values of C_i. Additionally, we define another bound lim_i as the highest value of the n minimum sum values.

In the next step, we find maximum valued lb_i and lim_i where $1 \leq i \leq n$, and call them lb_{max} and lim_{max}, respectively. Then we check if ub_i is smaller than lim_{max} for every i where $1 \leq i \leq n$. If it does so, then we remove the corresponding column from FM. The same procedure could be applied using lb_{max}. Using lb_{max} would allow us to remove columns without any increase on delay values. However, since using lim_{max} offers considerably higher number of columns to be removed with very small delay increases, we prefer to use it.

A pseudo code of our reducing algorithm is given in Algorithm 1. To elucidate the algorithm, an example is given in Figure 7. Here, 6×6 sized FM and VM are used. Considering C_1, it has four 1's. In order to find $n = 6$ maximum sum values, sum of the 4 highest values in each VM column is calculated. The highest value of these 6 sums is ub_1, which is found as 286 for this example. Similarly, while finding n minimum sum values, sum of the 4 lowest values in each VM column is calculated. Lowest value of these 6 minimum sums is lb_1 and the highest value is lim_1, which are respectively found as 111 and 210 for this example. After finding all ub_i, lb_i and lim_i values, lb_{max} and lim_{max} values are determined as 188 and 270. Then, ub_i of each C_i is compared with lb_{max} or lim_{max} to determine the columns to be removed. Note that using lim_{max} as bound results in more reduced columns (C_2, C_4, C_6) compared to using lb_{max} with two reduced columns (C_4, C_6).

3.2 Initial Column Mapping

We treat each of the columns of FM one by one starting from the column having the highest number of 1's to the lowest one. For each column of FM, first lb_i is determined in a similar way that we did in the matrix reducing step. The only difference is that, here we determine lb_i among unmapped columns of VM as opposed to considering all of the columns. Then, the VM column corresponding to lb_i is mapped to the column of FM. Algorithm 2 gives a pseudo code for the proposed column mapping.

Algorithm 2 Proposed Initial Column Mapping.

Input: $FM_{m\times n}$, $VM_{m\times n}$, column count n
Output: $FM_{initial}$
$FM_{sorted} \leftarrow$ high to low sort of FM columns for '1' sums
for each FM_{sorted} column i **do**
  $T_i \leftarrow$ transistor count of C_i
  for each VM column j **do**
    $All\_lb_{i,j} \leftarrow$ sum of the lowest T_i values on $VM(j)$
  end for
  $Pos \leftarrow$ position of minimum All_lb_i for C_i
  $FM_{initial}(Pos) \leftarrow C_i$
  remove $VM(Pos)$ column
end for

3.3 Hill Climbing Row Search with Column Reordering

After determining an initial column map, our algorithm starts its search by finding a PM column having the worst-case delay value, corresponding to the highest value in FPM. Then, in each try, two FM rows are interchanged such that the delay value of this worst-case column is reduced with a maximum amount, followed by a repeat of finding the worst-case PM column. The maximum amount is satisfied by finding the highest and lowest values in the VM column corresponding to 1 and 0 values in the FM column, respectively; the corresponding FM rows are interchanged. This whole procedure is constantly repeated until the new worst-case value is not smaller than the previous one anymore. If the new worst-case value is only getting larger, then the row search is stopped and the column reordering starts with considering the previous case having the most reducible worst-case delay.

After reaching a non-optimizable point, the algorithm checks which column has the worst-case delay value for most of the times on the previous row search step. This

column is switched with a column having the lowest transistor count. After each column reordering, the algorithm goes back to the row search. The number of column reorderings is upper limited by the number of columns. At the end, among all non-optimizable points, the best one is selected as the output.

A pseudo code of the row search and column reordering steps is given in Algorithm 3. To elucidate the algorithm, an example is given in Figure 8 for a 12×12 sized array. Here the point 'A' represent an initial column map. Each downward slope shows that the hill climbing row search is in duty. Column reorderings are shown with 'B' points. Finally, the point 'C' having the smallest delay among all non-optimizable points, is selected as the output.

Algorithm 3 Proposed Hill Climbing Row Search with Column Reordering.

Input: IMV, OMV, FM, VM, column count n, row count m
Output: $D_{final}, OMV_{final}, IMV_{final}$
$t \leftarrow 0$
while $t < n$ **do**
  $next \leftarrow 0$
  $D_{old} \leftarrow$ worst-case delay for IMV, OMV, VM set
  $P_{old} \leftarrow D_{old}$ column position
  while $next \neq 1$ **do**
    $P_{ones} \leftarrow$ '1' positions on P_{old} column, high to low
    $P_{zeros} \leftarrow$ '0' positions on P_{old} column, low to high
    for each index of P_{ones} i **do**
      for each index of P_{zeros} j **do**
        $IMV_{cur} \leftarrow$ switch i and j on IMV
        $D_{cur} \leftarrow$ worst-case delay for IMV_{cur}
        $P_{cur} \leftarrow D_{cur}$ column position
        if $D_{cur} < D_{my}$ **then**
          $D_{old} \leftarrow D_{cur}$
          $P_{old} \leftarrow P_{cur}$
          $IMV \leftarrow IMV_{cur}$
          add P_{cur} to P_{data} matrix
          break for loops
        end if
      end for
    end for
    if $D_{cur} > D_{old}$ **then**
      $next \leftarrow 1$
    end if
  end while
  add D_{old} to D_{data} matrix
  add OMV to OMV_{data} matrix
  add IMV to IMV_{data} matrix
  $t \leftarrow t + 1$
  if $t \neq n$ **then** ▷ Column Reordering
    $C_{wc} \leftarrow$ the most repeated column on P_{data}
    $M \leftarrow$ mean '1' count on FM column sums
    $C_{sorted} \leftarrow$ column low to high order for '1' sums
    $OMV \leftarrow$ switch C_{wc} and $C_{sorted}(mod(t, M))$
  end if
end while
$D_{final} \leftarrow$ minimum of D_{data}
$P_f \leftarrow$ position of D_{final} on D_{data}
$OMV_{final} \leftarrow P_f$th index of OMV_{data}
$IMV_{final} \leftarrow P_f$th index of IMV_{data}

3.4 Probabilistic Analysis of the Proposed Algorithm

Our proposed hill climbing algorithm uses binary switching between 0's and 1's of the current worst-case delay column.

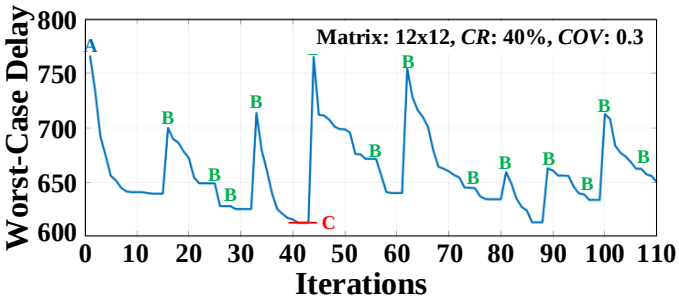


Fig. 8. An example for the proposed row search with column reordering processes.

If there is no better worst-case delay on the whole array after all binary moves, algorithm considers current state as a limit and conducts a column reordering to start search with a new column layout. Here, we can generate a fundamental probabilistic model to inspect algorithm characteristic.

There are two cases causing a stuck on a search. First, binary changes on the current worst-case column do not find a better worst-case on that particular column. Second, there is no better worst-case delay on the whole array after checking all binary changes. Inspecting the first case, it is apparent that this can happen only if all the crosspoint delays on 0 locations are higher than those on 1 locations. Given that the number of 1's in a column is P_1, we can find the number of 0's as $r - P_1$ where r is the row count. To generalize for random arrays, we can estimate P_1 as $r \times CR$ where CR is the crosspoint ratio. Therefore, probability of this specific failing ordering happening P_{Stuck} can be calculated as Equation 6. Note that $CR = 0.5$ gives the lowest value for this equation for any row count.

$$P_{Stuck} = \frac{(r \times CR)! \times (r - (r \times CR))!}{r!} \tag{6}$$

For the second case, since we do have any information about other columns rather than being lower than the inspected worst-case column, we can expect random outcomes for these columns on each binary row switch. Here, the successful case means finding a lower worst-case delay than that corresponding to the current worst-case column. We represent the current worst-case delay value as wc as a condition value to each column's probabilistic outcome. To reach a successful binary row search, all of the columns have to be lower than wc. This probability calculation is given in Equation 7.

$$P_{Continue} = \prod_{i=1}^{c-1} P_i(X < wc) \tag{7}$$

In conclusion, we can say that our algorithm fails either with a probability of P_{Stuck} or with a probability of $(1 - P_{Stuck})(1 - P_{Continue})$ as given in Equation 8. Note that finding better wc values means lower $P_{Continue}$ which result as higher P_{Fail} after each search step.

$$P_{Fail} = P_{Stuck} + (1 - P_{Stuck}) \times (1 - P_{Continue}) \tag{8}$$

From Equation 8, we can say that for a constant row count, increasing column count always results in a higher P_{Fail} since there would be more P_i products that decrease $P_{Continue}$ value while having constant P_{stuck}. This

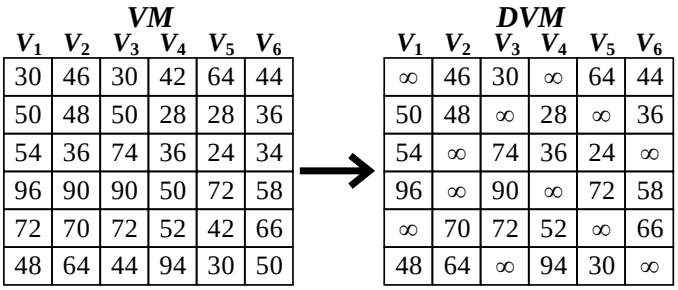


Fig. 9. DVM generation from VM by representing random defects as infinite delays.

means faster reach to the the lowest wc value to finish row search and conduct a column reorder, which result as lower runtimes but higher wc values. On the other hand, for a constant column count, increasing row count means lower P_{stuck} value, higher wc value, and lower coefficient of variation value on each column. For considerably high row counts, we expect column distributions to separate from each other, therefore resulting as higher $P_{continue}$ values. These inferences are justified by simulations in Section 4.

3.5 Proposed Algorithm on Defect and Variation Tolerance

As an extension for our hill climbing algorithm, we consider defect and variation tolerant logic mapping (DVTLM) problem. For this purpose we only update VM as DVM by assigning relatively high delay values, at least 100 times larger than the delay value on 3σ, to defective crosspoints. An example of DVM is given in Figure 9 with representing defects as infinite delays. Another note is that for the DVTLM problem, the proposed matrix reducing approach, as the first step of the proposed algorithm, can not be used since all defects spreading to the whole crossbar should be tolerated.

4 EXPERIMENTAL RESULTS

We present simulation results of our hill climbing algorithm for both VTLM and DVTLM problems. To generate FM's we use standard benchmarks from [27] as well as randomly generated benchmarks. For randomly generated benchmarks we use $CR = 40\%$, regarding that the average CR value for standard benchmarks is around 40%. To generate VM's, we assume that each crosspoint, and correspondingly each delay value in the matrix, shows an independent Gaussian (Normal) distribution with a mean μ, a standard deviation σ, and a coefficient of variation $COV = \sigma/\mu$. For further evaluations, we also consider different distributions including Weibull, exponential, and Beta distributions.

We mainly use COV values of 0.2 that is a common practice in the literature [23], [17], and [15], also supported by the reports of "International Technology Roadmap for Semiconductors (ITRS)" [28] and [29]. We also try different COV values between 0 and 0.3. For the DVTLM problem, we use defect rates between 5% and 40% independently for each crosspoint. We define a performance parameter "Delay Optimization Rate" as an improvement percentage from the initial worst-case delay value which occurs at random mapping.

We select a sample size of 2000 around which average runtime and delay values become steady. All simulations are conducted in MATLAB with a sample size of 100 around which average runtime and delay values become steady. All simulations run on a 3.50GHz Intel Core i5 CPU (only single core used) with 16GB memory.

4.1 Simulations for VTLM

We consider three state-of-the-art algorithms for comparison. The first one is the memetic algorithm given in [23]. The second one is the simulated annealing algorithm given in [18]. As the third, we develped a basic genetic algorithm which is inspired from [19] and [22]. The source code of these algorithms as well as our proposed algorithm with supporting materials are available at http://www.ecc.itu.edu.tr/images/a/a0/VTLM.zip

Before presenting time and delay values of our algorithm, we evaluate the first two steps of the algorithm called matrix reducing and initial column mapping. Note that both of these steps are algorithm independent, so they can be applied to any VTLM algorithm. We apply our matrix reducing method to three different algorithms; results are given in Table 1. Since the memetic algorithm in [23] is specifically designed for the DVTLM problem, it is not suitable for matrix reducing. We see that using reduced matrices provides us an important time saving, up to 72% at the cost of slight increase in the worst-case delay up to 3%. Note that these simulations are done using an upper limit of lim_{max}, previously defined in Subsection 3.1. If we used lb_{max} instead of lim_{max}, then delay increase would not happen at all. However, time decrease rate would be lower.

We evaluate the proposed initial column mapping technique by comparing it with random mapping and greedy mapping method proposed in [20]; all of these methods are used with our hill climbing algorithm. Randomly generated benchmarks with different sizes are used for a fair comparison. Results as delay optimization rates are given in Table 2. We see that the proposed technique always gives the best result, which is 3-5% better than greedy from the literature [20] and 10-15% than random mapping. The reason is that the greedy mapping proposed in [20], and also used in [22], determines lb_i's by using sums of all delay values in a VM column. However, we select a minimum sum of T_i delay values where T_i is the number of 1's in the FM column to be mapped.

In Figure 10 we compare the runtimes of the initial column mapping step and the matrix reducing step with the hill climbing row search plus column reordering steps of the algorithm. We see that, the runtime overhead of the initial column mapping and matrix reducing steps is quite low, always smaller than 5% and gets dramatically smaller as the matrix size increases.

The performance of our hill climbing algorithm is summarized in Figure 11. Recall that the overwhelming proportion of the algorithm's computational load corresponds to the step of row search and column reordering. Since these steps get into each other, a similar runtime behaviour is expected for the changes in the row and column counts. This is shown in Figure 11 (a). However, for the delay

TABLE 1
Delay increase and time decrease rates after applying the proposed matrix reducing technique; $COV = 0.2$.

		Proposed Hill Climbing		Simulated Annealing [18]		Genetic Alg. [19], [22]	
Benchmark $(C \times R)$	**Reduced Column Ratio**	**Delay Increase Rate**	**Time Decrease Rate**	**Delay Increase Rate**	**Time Decrease Rate**	**Delay Increase Rate**	**Time Decrease Rate**
5ex1 (75×14)	71%	0.2%	64.8%	1.3%	32.26%	1.8%	4.5%
Inc (34×14)	39%	2.6%	27.2%	0.1%	26.2%	0.5%	8.6%
clip (167×18)	19%	0.6%	24.9%	0.3%	15.6%	0.1%	3.5%
Misex2 (29×40)	72%	0.2%	76.4%	1.1%	26.8%	0.1%	4.48%
9sym (87×18)	-	-	-	-	-	-	-
Bw (65×10)	%44.3	3.1%	41.08%	1.4%	19.02%	0.6%	2.9%
Rd53 (32×10)	4%	1.03%	10.2%	1.2%	1.56%	0.4%	3.05%
Rd73 (141×14)	24%	0.7%	22.8%	1.12%	17.43%	0.8%	0.5%
Sao2 (58×18)	48.2%	1.05%	24.9%	1.3%	19.86%	0.3%	2.4%
Table5 (158×34)	44.2%	1.2%	50.6%	0.4%	34.12%	0.3%	22.6%

TABLE 2
Delay optimization rates for the proposed hill climbing algorithm having different initial column mapping methods.

Benchmark $(C \times R)$ **(CR: 40%)**	**Proposed Greedy**	**Greedy [20]**	**Random**
Random 1 (6×6)	21.8%	20.44%	17.45%
Random 2 (12×12)	21.82%	21.01%	17.96%
Random 3 (24×24)	22.08%	21.3%	19.21%
Random 4 (48×48)	20.84%	20.57%	18.63%

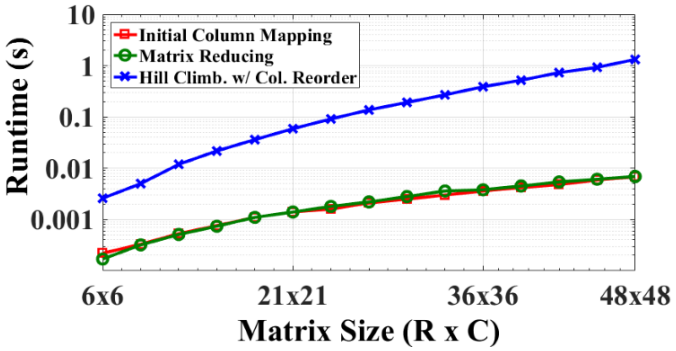


Fig. 10. Runtime comparison of the initial column mapping step, the matrix reducing step, and the hill climbing row search with column reordering steps for different matrix sizes on logarithmic time scale ($COV = 0.2$, $CR = 40\%$).

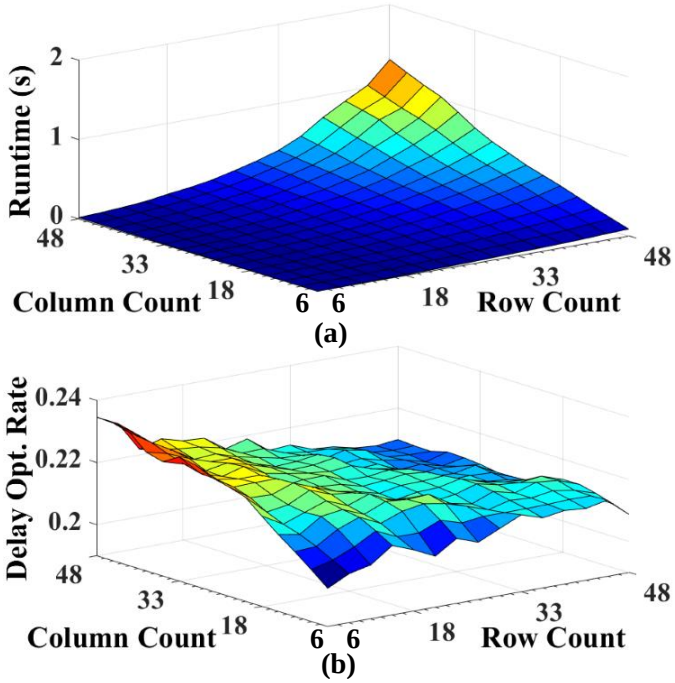


Fig. 11. (a) Runtimes and (b) delay optimization rates of the proposed hill climbing algorithm for different sized random arrays ($COV = 0.2$, $CR = 40\%$).

values, row and column counts have different effects as shown in Figure 11 (b). Since delays are calculated column-wise and correspondingly our algorithm aims to minimize column delays, change in the number of columns affects delay optimization rates more than the row change does. Also, high row count converges to 20% delay optimization rate because of the decreasing COV values of the columns sums, since mean 'μ' values multiplies with the '1' count while standart deviation 'σ' values multiplies with the root of the '1' count. This property of Gaussian sums results as more separated column delay distributions as the row count increases. As a result, there are relatively less columns to be optimized, which results as effective and similar results.

We also make detailed comparisons of our algorithm with three different algorithms in the literature in Table 3. In terms of the worst-case delay, represented by "Delay Opt. Rate", our algorithm gives the best results in almost all cases. Here, our algorithm has similar optimization rate results, around 20%, for all random array sizes. This proves our algorithms' scalability for large logic functions since other algorithm results get worse as the matrix size increases. Also, in terms the runtime, our algorithm gives by far the best results in all cases. Note that we run the memetic algorithm for at most 20 seconds. The reason is that if the algorithm does not give any solution for this time span, then it is unlikely to get results in practical time limits. Note that we apply initial column mapping on all of these algorithms for a fair comparison.

In the following two subsections, we further evaluate our algorithm for different variance values, and for different distributions.

4.1.1 *Evaluation for Different Variance Values*

Depending on the manufacturing technology, COV values might change. Future technologies might have lower or higher variations with better manufacturing costs. Therefore, we inspect performance of our hill climbing algorithm

TABLE 3
Delay optimization rate and runtime comparisons; $COV = 0.2$.

Benchmark ($C \times R$)	Hill Climbing		Memetic Alg.		Simulated Ann.		Genetic Alg.	
	Delay Opt. Rate	Runtime (s)	Delay Opt. Rate	Runtime (s)	Delay Opt. Rate	Runtime (s)	Delay Opt. Rate	Runtime (s)
5ex1 (75×14)	**25.7%**	**0.11**	-	>20	15.3%	0.185	22%	0.69
inc (34×14)	**20.8%**	**0.0096**	20.6%	5.4	15%	0.67	14.5%	0.76
clip (167×18)	**19.01%**	**0.09**	-	>20	9.3%	0.18	11.2%	0.12
Misex2 (29×40)	**24.5%**	**0.094**	15.2%	1.4	13%	0.97	11.1%	0.54
9sym (87×18)	**12.5%**	**0.072**	-	>20	8.3%	0.11	8.8%	0.78
Bw (65×10)	**20.8%**	**0.0078**	-	>20	13.7%	0.15	12.6%	0.74
Rd53 (32×10)	19.2%	**0.0086**	**23.1%**	7.1	11.6%	0.12	10.1%	0.34
Rd73 (141×14)	**13.86%**	**0.038**	-	>20	7.8%	0.27	8.1%	0.82
Sao2 (58×18)	**17.94%**	**0.1**	-	>20	8.9%	0.16	9%	0.78
Table5 (158×34)	**16.1%**	**0.55**	-	>20	7.5%	0.59	8.3%	1.85
6×6 ($CR = 40\%$)	**21.8%**	**0.0007**	20.3%	1.02	20.68%	0.04	10.9%	0.53
12×12 ($CR = 40\%$)	**21.82%**	**0.0025**	20.6%	1.03	17.13%	0.4	19%	0.61
24×24 ($CR = 40\%$)	**22.08%**	**0.015**	16.1%	1.2	13.05%	0.68	11.5%	0.78
48×48 ($CR = 40\%$)	**20.84%**	**0.37**	10.8%	4.24	10.92%	1.45	15.3%	1.14

*Bold values/elements represent the best results

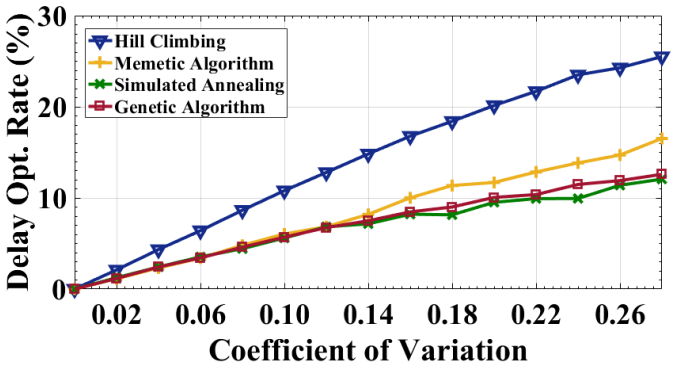


Fig. 12. For COV values between 0 and 0.3, delay optimization rates for different algorithms on 48×48 matrix with $CR = 40\%$.

along with the other algorithms for different COV values ranging from 0 to 0.3. Results are given in Figure 12. We see that our proposed algorithm scales much better than the other search algorithms as COV value increases.

4.1.2 Evaluation for Different Distribution Types

To inspect different possible manufacturing properties we inspect different distributions. Along with a symmetrical Gaussian distribution, we use Weibull and Beta distributions with slightly skewed curves of density functions. Also we use an exponential distribution having extremely skewed density function curve. All distributions have $COV = 0.2$ to preserve a fair comparison. Results are given in Table 4. Here, compared to Gaussian distribution, slight differences on optimization rates occur on Weibull and Beta distributions. However, exponential distribution has much higher rates for each algorithm due to having extreme worst-case delays far from mean values. Examining the results we see that our algorithm's superiority for Gaussian distribution is more less valid for the other distributions.

4.2 Simulations for DVTLM

We define "Success Rate" as the ratio of cases or samples for which all defects are tolerated to the total number of cases. Figure 13 and Figure 14 shows success rate or our algorithm for different defect rates and for different benchmarks. Figure 13 tells us that increasing defect rate beyond 10% dramatically worsens the success rate, but depending on the matrix structure 20% defect rate might be tolerated with a high success rate. On the Figure 14 we campared two cases; constant column count with the increasing row count in Figure 14 (a) and and vice-versa in Figure 14 (b). Here, increasing row count doesn't change defect tolerance effectiveness, since our algorithm excels on row search. On the other hand, increasing column count decrease defect tolerance effectiveness since we have to tolerate more column formations with limited row switching action.

TABLE 4
Delay optimization rate comparison of different distribution types for different search algorithms with different sized matrices; $COV = 0.2$, $CR = 40\%$.

Algorithms	Matrix Size	Gaussian Dist.	Weibull Dist.	Beta Dist.	Exp. Dist.
Proposed Hill Climbing	6×6	19.01%	18.3%	19.46%	43.1%
	12×12	20.3%	19.1%	22.07%	44.92%
	24×24	20.9%	19.5%	22.53%	43.01%
	48×48	20.14%	19%	20.77%	37.8%
Memetic Algorithm	6×6	19.61%	19.02%	20.53%	45.65%
	12×12	20.54%	19.47%	20.74%	44.13%
	24×24	16.39%	16.48%	16.89%	37.12%
	48×48	12.37%	12.31%	12.96%	33.26%
Simulated Annealing	6×6	18.7%	17.7%	20.4%	41.2%
	12×12	14.68%	14.3%	19.06%	30.1%
	24×24	10.6%	10.8%	14.61%	21.2%
	48×48	7.71%	7.72%	10.97%	15.09%
Genetic Algorithm	6×6	10%	10.3%	9.3%	29.2%
	12×12	10.9%	11.7%	10.8%	28.45%
	24×24	11.1%	11.6%	11.3%	27.88%
	48×48	10.1%	10.4%	9.5%	21.2%

Table 5 and Table 6 give detailed comparisons of our algorithm with three different algorithms for 5% and 10% defect rates, respectively. As expected, increase in defect rate not only decrease success rate, but also worsens variation tolerance performance. Examining the numbers, we see that for overwhelming majority of cases, our algorithm gives the best result. Also, results approves the superiority of our algorithm's speed.

TABLE 5
Success rate, delay optimization rate, and runtime comparisons; 5% defect rate, $COV = 0.2$.

Defect Rate: 5%	Proposed Hill Climbing			Memetic [23]			Simulated Annealing [18]			Genetic [19], [22]		
Benchmark ($C \times R$)	Succ. Rate	Delay Opt. Rate	Run time	Succ. Rate	Delay Opt. Rate	Run time	Succ. Rate	Delay Opt. Rate	Run time	Succ. Rate	Delay Opt. Rate	Run time
5ex1 (75×14)	**96%**	16.6%	**0.059**	-	-	>20	0%	-	0.16	5%	**21.2%**	0.75
inc (34×14)	**100%**	16.4%	**0.032**	**98%**	**18.9%**	1.7	53%	5.5%	1.25	43%	14.3%	0.6
clip (167×18)	**2%**	**8.6%**	**0.29**	-	-	>20	0%	-	0.67	0%	-	1.07
Misex2 (29×40)	**100%**	**24%**	**0.2**	**100%**	15.3%	1.22	23%	4.8%	0.33	79%	12%	0.82
9sym (87×18)	**25%**	**5.1%**	**0.146**	-	-	>20	0%	-	0.39	0%	-	0.77
Bw (65×10)	**85%**	**14%**	**0.027**	-	-	>20	38%	4.2%	0.29	12%	12.4%	0.6
Rd53 (32×10)	96%	12.2%	**0.014**	**100%**	**21.7%**	2.95	56%	5.5%	0.22	43%	10.4%	0.55
Rd73 (141×14)	**1%**	**6.6%**	**0.14**	-	-	>20	0%	-	0.5	0%	-	0.81
Sao2 (58×18)	**77%**	**11.1%**	**0.084**	-	-	>20	0%	-	0.32	1%	8.3%	0.69
Table5 (158×34)	0%	-	3.01	-	-	>20	0%	-	0.99	0%	-	1.61
6×6 ($CR = 40\%$)	**100%**	**21%**	**0.0027**	**100%**	17.9%	1.03	**100%**	19.3%	0.02	99%	14.3%	0.52
12×12 ($CR = 40\%$)	**100%**	**18.61%**	**0.006**	**100%**	18.2%	1.05	**100%**	14.7%	0.3	93%	16.8%	0.59
24×24 ($CR = 40\%$)	**100%**	**17.6%**	**0.09**	**100%**	15.07 %	1.15	6%	6.05%	0.42	38%	13.09%	0.76
48×48 ($CR = 40\%$)	**100%**	**18.9%**	**1.8**	**100%**	10.2%	2.24	0%	-	1.25	0%	-	1.13

*Bold values/elements represent the best results

TABLE 6
Success rate, delay optimization rate, and runtime comparisons; 10% defect rate, $COV = 0.2$.

Defect Rate: 10%	Proposed Hill Climbing			Memetic [23]			Simulated Annealing [18]			Genetic [19], [22]		
Benchmark ($C \times R$)	Succ. Rate	Delay Opt. Rate	Run time	Succ. Rate	Delay Opt. Rate	Run time	Succ. Rate	Delay Opt. Rate	Run time	Succ. Rate	Delay Opt. Rate	Run time
5ex1 (75×14)	**2%**	**15.7%**	**0.056**	-	-	>20	0%	-	0.17	0%	**-**	0.68
inc (34×14)	26%	10.8%	**0.03**	**82%**	**17.3%**	1.51	0%	-	1.29	1%	14.1%	0.608
clip (167×18)	0%	**-**	0.32	-	-	>20	0%	-	0.69	0%	-	1.14
Misex2 (29×40)	**100%**	**22.1%**	**0.23**	**100%**	12.9%	1.11	0%	-	0.35	2%	7.1%	0.827
9sym (87×18)	0%	**-**	0.142	-	-	>20	0%	-	0.41	0%	-	0.76
Bw (65×10)	**14%**	**12.1%**	**0.02**	-	-	>20	0%	-	0.3	0%	-	0.6
Rd53 (32×10)	18%	8.4%	**0.014**	**98%**	**20.6%**	2.11	0%	-	0.22	0%	-	0.56
Rd73 (141×14)	0%	**-**	0.14	-	-	>20	0%	-	0.51	0%	-	0.8
Sao2 (58×18)	**1%**	**3.4%**	**0.09**	-	-	>20	0%	-	0.34	0%	-	0.69
Table5 (158×34)	0%	-	3.03	-	-	>20	0%	-	1.02	0%	-	1.81
6×6 ($CR = 40\%$)	**100%**	14%	**0.0017**	**100%**	**18.2%**	1.04	95%	13.7%	0.15	97%	17.5%	0.52
12×12 ($CR = 40\%$)	**100%**	14.4%	**0.008**	**100%**	**19.06%**	1.02	**94%**	13.6%	0.32	79%	14.2%	0.59
24×24 ($CR = 40\%$)	97%	**17.14%**	**0.091**	**100%**	14.7%	1.13	0%	-	0.44	0%	-	0.67
48×48 ($CR = 40\%$)	0%	**-**	2.03	0%	-	2.2	0%	-	1.36	0%	-	1.13

*Bold values/elements represent the best results

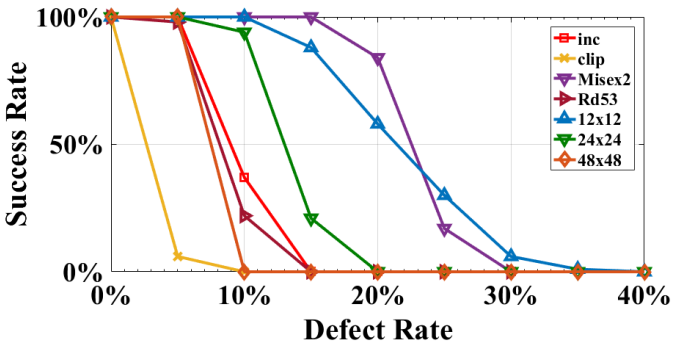


Fig. 13. Success rate of our hill climbing algorithm for different benchmarks with $CR = 40\%$.

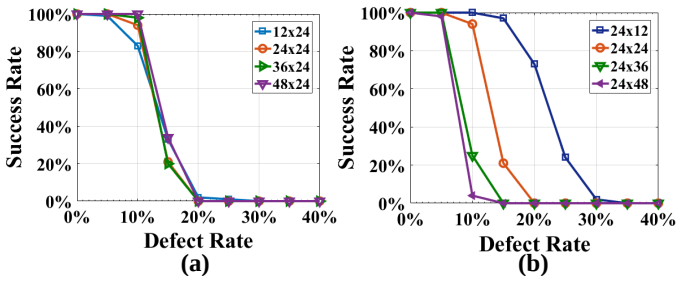


Fig. 14. Success rate of our hill climbing algorithm with increase in (a) row, and (b) column size; $CR = 40\%$.

5 CONCLUSION

In this work, we propose a variation tolerant logic mapping algorithm to optimize the worst-case delay of nano-crossbars. We show that our algorithm can be successfully used for defect tolerance, so defect and variation tolerance can be achieved at the same time. Simulations show that our algorithms runs considerably faster than the previously proposed algorithms with offering similar or better delay improvements. Difference between delay optimization rates of our algorithm and reference algorithms increases as the matrix gets larger, which proves that our algorithm has better scalability for real-world applications.

The proposed algorithm is technology independent that can be used for any technology using PLA like computing as well as for conventional CMOS PLA circuits. As a future work, we consider modifying and improving this algorithm for cascaded arrays to adapt real-life applications. Also, we plan to extend this work to be applicable for transient variations due mainly to degradations occurred in crossbars.

Relatively, area yield optimizations can be performed.

REFERENCES

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

[2] A. Zhang, G. Zheng, and C. M. Lieber, "Nanoelectronics, circuits and nanoprocessors," in *Nanowires*. Springer, 2016, pp. 103–142.

[3] D. Alexandrescu, M. Altun, L. Anghel, A. Bernasconi, V. Ciriani, L. Frontini, and M. Tahoori, "Synthesis and performance optimization of a switching nano-crossbar computer," in *Digital System Design (DSD), 2016 Euromicro Conference on*. IEEE, 2016, pp. 334–341.

[4] M. M. Ziegler and M. R. Stan, "Cmos/nano co-design for crossbar-based molecular electronic systems," *IEEE Transactions on Nanotechnology*, vol. 2, no. 4, pp. 217–230, 2003.

[5] Z. Zhong, D. Wang, Y. Cui, M. W. Bockrath, and C. M. Lieber, "Nanowire crossbar arrays as address decoders for integrated nanosystems," *Science*, vol. 302, no. 5649, pp. 1377–1379, 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] 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.

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

[9] G. Snider, P. Kuekes, T. Hogg, and R. S. Williams, "Nanoelectronic architectures," *Applied Physics A*, vol. 80, no. 6, pp. 1183–1195, 2005.

[10] A. DeHon and B. Gojman, "Crystals and snowflakes: building computation from nanowire crossbars," *Computer*, vol. 44, no. 2, pp. 37–45, 2011.

[11] H. Hamoudi, "Crossbar nanoarchitectonics of the crosslinked self-assembled monolayer," *Nanoscale research letters*, vol. 9, no. 1, p. 287, 2014.

[12] M. C. Morgul, F. Peker, and M. Altun, "Power-delay-area performance modeling and analysis for nano-crossbar arrays," in *VLSI (ISVLSI), 2016 IEEE Computer Society Annual Symposium on*. IEEE, 2016, pp. 437–442.

[13] 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.

[14] 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.

[15] 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.

[16] O. Tunali and M. Altun, "A survey of fault-tolerance algorithms for reconfigurable nano-crossbar arrays," *ACM Comput. Surv.*, vol. 50, no. 6, pp. 79:1–79:35, Nov. 2017. [Online]. Available: http://doi.acm.org/10.1145/3125641

[17] B. Gojman and A. DeHon, "Vmatch: Using logical variation to counteract physical variation in bottom-up, nanoscale systems," in *Field-Programmable Technology, 2009. FPT 2009. International Conference on*. IEEE, 2009, pp. 78–87.

[18] C. Tunc and M. B. Tahoori, "Variation tolerant logic mapping for crossbar array nano architectures," in *Proceedings of the 2010 Asia and South Pacific Design Automation Conference*. IEEE Press, 2010, pp. 855–860.

[19] Y. Yang, B. Yuan, and B. Li, "Defect and variation tolerance logic mapping for crossbar nanoarchitectures as a multi-objective problem," in *Information Science and Technology (ICIST), 2011 International Conference on*. IEEE, 2011, pp. 1139–1142.

[20] F. Zhong, B. Yuan, and B. Li, "Hybridization of nsga-ii with greedy re-assignment for variation tolerant logic mapping on nano-scale crossbar architectures," in *Proceedings of the Companion Publication of the 2014 Annual Conference on Genetic and Evolutionary Computation*. ACM, 2014, pp. 97–98.

[21] 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," *IEEE Transactions on Evolutionary Computation*, vol. 18, no. 6, pp. 846–859, 2014.

[22] F. Zhong, B. Yuan, and B. Li, "A hybrid evolutionary algorithm for multiobjective variation tolerant logic mapping on nanoscale crossbar architectures," *Applied Soft Computing*, vol. 38, pp. 955–966, 2016.

[23] B. Yuan, B. Li, H. Chen, and X. Yao, "Defect-and variation-tolerant logic mapping in nanocrossbar using bipartite matching and memetic algorithm," *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, vol. 24, no. 9, pp. 2813–2826, 2016.

[24] M. Zamani and M. B. Tahoori, "Variation-aware logic mapping for crossbar nano-architectures," in *Proceedings of the 16th Asia and South Pacific Design Automation Conference*. IEEE Press, 2011, pp. 317–322.

[25] B. Ghavami, A. Tajary, M. Raji, and H. Pedram, "Defect and variation issues on design mapping of reconfigurable nanoscale crossbars," in *VLSI (ISVLSI), 2010 IEEE Computer Society Annual Symposium on*. IEEE, 2010, pp. 173–178.

[26] B. Ghavami, "Joint defect-and variation-aware logic mapping of multi-outputs crossbar-based nanoarchitectures," *Journal of Computational Electronics*, vol. 15, no. 3, pp. 959–967, 2016.

[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.

[28] "International technology roadmap for semiconductors," https://www.semiconductors.org/main/2009_international_technology_roadmap_for_semiconductors_itrs/, 2009.

[29] B. Hoefflinger, "Itrs: The international technology roadmap for semiconductors," in *Chips 2020*. Springer, 2011, pp. 161–174.

Furkan Peker recieved his BSc in Yıldız Technical University and MSc degree in İstanbul Technical University in 2014 and 2017 respectively. He finished his BSc project with research scholarship from Scientific and Technological Research Council of Turkey (2241/A) in 2014 with a success and graduated as honoured student. He finished another Scientific and Technological Research Council of Turkey project as Synthesis and Reliability Analysis of Nano Switching Arrays with Mustafa Altun in 2017 and published a conference paper based on this project. He is currently a research assistant in İstanbul Technical University in İstanbul since 2014.

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.