An Island Style Multi-Objective Evolutionary Framework for Synthesis of Memristor-Aided Logic

Umar Afzaal
School of EE, KAIST
Daejeon, Korea
engr.umarafzal@gmail.com

Seunggyu Lee
School of EE, KAIST
Daejeon, Korea
sg.lee@kaist.ac.kr

Youngsoo Shin
School of EE, KAIST
Daejeon, Korea
youngsoo@kaist.edu

ABSTRACT

The optimal in-memory mapping onto memristor crossbars involves competing design goals: minimizing crossbar utilization, reducing delay, and achieving an even layout. Existing heuristic algorithms struggle to address these objectives simultaneously, often yielding suboptimal solutions. This paper introduces an automatic design solution to optimize multiple objectives concurrently. Specifically, it proposes an island-style evolutionary algorithm for multi-objective optimization of in-memory mapping. This algorithm produces a set of solutions, corresponding to Pareto points. Each point can be stored in a library of mapping solutions, which can be chosen when corresponding design is re-used as a macro. Experimental evaluation on IWLS benchmarks demonstrates the effectiveness of this approach in addressing multiple design objectives efficiently.

KEYWORDS

Memristor-aided logic, logic synthesis, evolutionary algorithm

ACM Reference Format:
Umar Afzaal, Seunggyu Lee, and Youngsoo Shin. 2025. An Island Style Multi-Objective Evolutionary Framework for Synthesis of Memristor-Aided Logic. In *30th Asia and South Pacific Design Automation Conference (ASPDAC '25), January 20–23, 2025, Tokyo, Japan.* ACM, New York, NY, USA, 7 pages. https://doi.org/10.1145/3658617.3697579

1 INTRODUCTION

The bottleneck in von Neumann computing systems stems from the imbalance between CPU and memory speed improvements. Traditionally, CPU speeds doubled annually, whereas memory speeds only did so every decade. Consequently, compute units can execute operations faster than memory units can load and store necessary data, shifting the primary time and energy consumption away from compute units. This problem is compounded by the limited bandwidth of buses, which becomes particularly problematic during large data transfers throttling application throughput and latency. Thus, the cost of transferring data between memory and compute units has emerged as the modern computing bottleneck. Note that data transfer from the processor to memory also has high a energy cost, with DRAM access using a three order of magnitude more energy than a 32-bit addition operation [2].

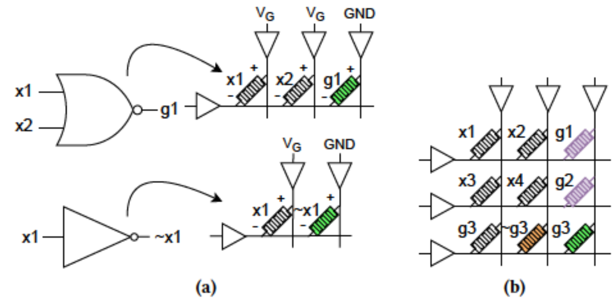

Figure 1: (a) NOR and INV implementation with memristors using MAGIC. (b) Parallel operations and copy operations.

The idea of processing-in-memory (PIM) is to realize logic operations directly in memory, decreasing data transfer needs. PIM accelerates tasks that involve repetitive operations on large datasets with heavy memory read/write activity. A prime example is the training and inference of machine learning (ML) models, which entails an enormous number of arithmetic operations such as additions, multiplications, or convolutions, depending on the model. In fact, the high energy demands of billion-parameter models have already become a serious concern [6]. PIM is positioned as a key solution going forward to meet the rising costs and performance demands of large-scale ML models.

The solution is to leverage devices that inherently support computation within memory arrays. Memristors which are the focus of this paper, characterized by their non-volatile resistive memory properties, stand out in this category. These two-terminal devices offer two resistance states, which can be switched by applying voltage, thereby reliably encoding logic 1 and 0 through low and high resistance states, respectively. The scope of this paper is the memristor-based logic design families proposed in recent years. Among them, the two leading logic design styles are IMPLY [4], which utilizes material implication operations with two memristors and a resistor, and more recently MAGIC [5], which constructs NOR/INVerter(INV) logic gates entirely from memristors. MAGIC is particularly advantageous for requiring only all-memristor crossbars and thus more suited for driving PIM technology of the future.

Under the rules of MAGIC, to implement either an INV or a NOR gate, each gate input and the output are stored in distinct memristors. Input values are pre-stored as initial resistances of the input memristors. As depicted in Figure 1(a), $x1$ and $x2$ are the inputs to a NOR gate. The NOR output is produced in the

Permission to make digital or hard copies of part or all of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for third-party components of this work must be honored. For all other uses, contact the owner/author(s).

ASPDAC '25, January 20–23, 2025, Tokyo, Japan

© 2025 Copyright held by the owner/author(s).

ACM ISBN 979-8-4007-0635-6/25/01.
https://doi.org/10.1145/3658617.3697579

memristor labelled $g1$. To prepare for computation, this memristor is initialized with logic-1 by applying a voltage that sets its resistance to R_{ON} (low resistance). The NOR operation proceeds by applying a gate voltage V_G to each input memristor and grounding the output memristor. This operation switches the state of the output memristor to logic-0 characterized R_{OFF} (high resistance) if either or both of the inputs held logic-1.

All memristors involved must be oriented either in a row or column, with order being irrelevant. A key characteristic of MAGIC is its ability to generate the same gate output across multiple memristors within the same cycle. It's also feasible to compute multiple gate outputs in the same clock cycle if all input memristors of each gate are aligned, along with their corresponding output memristors. This is illustrated in Figure 1(b), where outputs $g1$ and $g2$ are eligible for single-cycle computation. Copying a value from one memristor to another involves two INV operations, aptly called a copy operation, which completes in two cycles. As demonstrated in Figure 1(b), $g3$ is copied from $(2, 0)$ to $(2, 2)$. Please note that this primer on MAGIC is not self-contained and readers are encouraged to consult the original paper that introduced the MAGIC logic [5].

1.1 Motivational Example

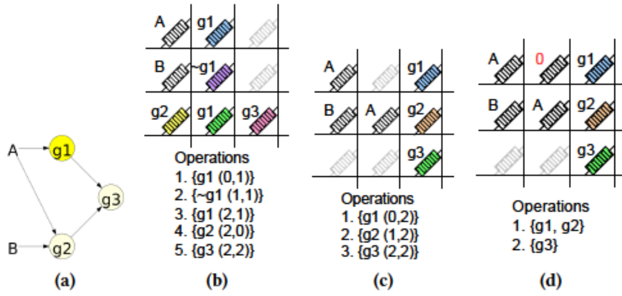

Figure 2: (a) Example NIG. Light yellow represents 2-input NOR gate. (b) Mapping synthesized with [1]. (c) Alternate mapping (d) Yet another mapping with reduced cycles.

Example 1: In the NOR-INV graph (NIG) illustrated in Figure 2(a), Gharpinde et al.'s heuristic [1] places primary inputs (PIs) in the first column, directs INV gates eastward, and NOR gates southward whenever possible, as detailed in Algorithm 2 of their article. This results in a 3×3 crossbar with 7 memristors used over 5 cycles for logic operations, as shown in Figure 2(b). In contrast, the mapping in Figure 2(c) uses only 6 memristors and completes operations in 3 cycles. This improvement is achieved by storing PI value A in two memristors, avoiding the need to copy $g1$'s value from (0,1) to (2,1). Additionally, Figure 2(d) presents a mapping treating the INV gate as a NOR with one input tied to logic-0. This approach uses 7 memristors but completes logic in just 2 cycles, marking a 60% reduction in clock cycles compared to the initial 5 cycles.

In addressing the mapping problem, key decision variables include gate location, orientation, the number of gate instances on the crossbar, and simultaneous gate output calculations. This problem is NP-complete, prompting various heuristic and optimization approaches in literature. Similar to Gharpinde et al. [1], the staircase-structure inspired mapping [9] aims to maximize parallel operations and crossbar squareness with a heuristic. This latter approach requires aligning copy operations for multiple gates, introducing some copy overhead. In contrast, optimization methods, such as the one described in [3], address latency through integer linear programming (ILP). Although effective, this approach can be time-consuming for larger circuits due to exhaustive exploration of operation possibilities. To address this, [8] suggests employing multiple ILPs by partitioning the netlist, although manual partitioning does not allow exploration of this region of the solution space. Lastly, these approaches typically yield a single solution.

1.2 Article Contributions

Genetic algorithms (GAs) have historically achieved significant success in various aspects of electronic design automation subject to optimization. This work applies GAs to the problem of MAGIC synthesis. The main contributions are summarized below:

- Optimization problem without manual partitions that enables finding the optimal grouping of gates for parallel computation, minimizing clock cycles.
- An algorithm that converts any invalid mapping to a legal one, following MAGIC rules.
- A distributed evolutionary framework to automatically map a NIG to a memristor crossbar, yielding a set of non-dominated solutions.

2 PROPOSED METHOD

We define the optimization problem by allowing only one instance per each gate output in the NIG. This fixes the number of decision variables subject to genetic operators. However, more than one instance of a gate may be required on the crossbar as well as the PIs for realizing a legal MAGIC circuit. To which end, for evaluation, this partial mapping is legalized by means of another algorithm presented in Section 2.3. Thus, decision variables are not subject to any constraints allowing any arrangement of all gate instances on the crossbar. The complete flow is shown in Figure 3.

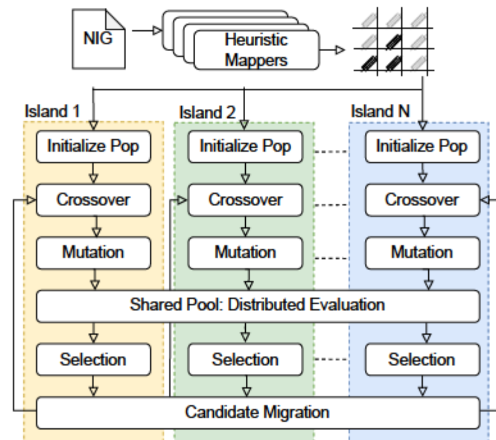

Figure 3: Overall flow of the proposed method.

The GA is implemented as an island model with multiple populations. Populations can be initialized with seeds acquired from

different heuristic mappers. To obtain a valid seed, the mapping should first be stripped to one instance per gate carefully. Seeding is optional and can be used if it encourages faster convergence to optimal solutions. The selection and variation operators are standard NSGA-II operators. For migration, a typical ring topology is employed where the number of emigrants is determined by the mig_rate hyperparameter while migration interval is controlled by $mig_interval$.

2.1 Point Mutation

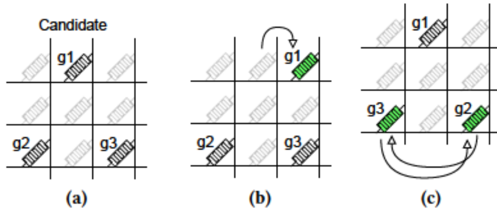

Figure 4: (a) Target mapping. (b) Gate relocation. (c) Gate swapping.

Mutation is conducted as follows: a random number of gates is selected from a Levy distribution $\mathcal{L}$. For each gate g_x, a random decision is made between relocating it in the same row or the same column, allowing the gate to move only in orthogonal steps. The step size is also chosen from the Levy distribution $\mathcal{L}$. If the new location is vacant, the gate is simply moved there. However, if the new location already contains a gate g_y, then a swap occurs between the two gates. Both of these scenarios are illustrated in Figure 4. The number of candidates to mutate in a population is determined by the hyperparameter $mutpb$.

2.2 Crossover

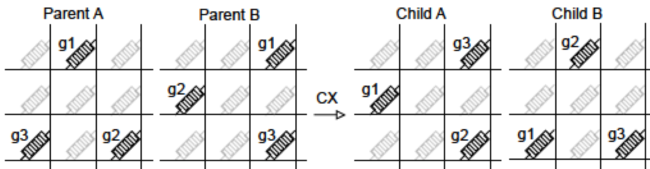

Figure 5: Showcasing the crossover operation.

During crossover, two parents combine to create two new children with shared genetic material. Let's denote the set of gates mapped onto the set of memristors M (the crossbar) of size $m \times n$ as G, then $g_{i,j} \in G \; for \; 0 \leq i < m, 0 \leq j < n$ represents a gate mapped at the ith row and j^{th} column. Then, the following two conditions are applied to extract genetic material for crossover:

$$\forall g \in G \mid (loc(g) \in M_1) \land (loc(g) \in M_2 = free) \quad (1)$$

$$\forall g \in G \mid (loc(g) \in M_2) \land (loc(g) \in M_1 = free) \quad (2)$$

Referring to M_1 as the set of memristors for the first parent and vice versa, from each parent, extract gates that, at the same locations in the other parent are $free$ (i.e., $free$ is a memristor onto which no gate output has been mapped). Applying to the example shown in Figure 5, these set of gates are extracted for each parent:

$$for \; ParentA: \; \{g1 : (0,1)\} \; \{g3 : (2,0)\}$$
$$for \; ParentB: \; \{g1 : (0,2)\} \; \{g2 : (1,0)\}$$

The gates' locations are exchanged such that gate $g1$ from $ParentA$, initially at $(0,1)$, receives the location of $ParentB'$s last gate, $g2$, making $g1$ at $(1,0)$ for the first child. Each gate adopts the location of the corresponding gate from the other parent in reverse order. The resulting children are given as:

$$ChildA: \; \{g1 : (1,0)\} \; \{g2 : (2,2)\} \; \{g3 : (0,2)\}$$
$$ChildB: \; \{g1 : (2,0)\} \; \{g2 : (0,1)\} \; \{g3 : (2,2)\}$$

2.3 Mapping Completion Algorithm

To evaluate a candidate's quality based on memristor utilization, crossbar size, and clock cycles, the gate placement obtained from genetic operators undergoes the following steps to convert it into a valid MAGIC circuit. These steps will be demonstrated by solving Example 2, where the NIG is shown in Figure 6 and the mapping is completed as depicted in Figure 7.

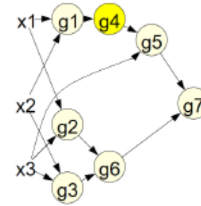

Figure 6: NIG for Example 2.

2.3.1 Find Aligned Gates. In the first step, gates aligned in rows and columns are grouped. For each row i, let's denote the set of columns with gates mapped onto memristors as H_i. Similarly, for each column j, let's denote the set of rows with gates mapped onto memristors as V_j. The conditions for H_i and V_j can be expressed as:

$$H_i = \{M_{i,j} \neq free\} \; \forall \; 0 \leq j < n \quad (3)$$

$$V_j = \{M_{i,j} \neq free\} \; \forall \; 0 \leq i < m \quad (4)$$

Conditions 3 and 4 are applied to each row i and each column j, respectively to obtain all groups of aligned gates. For the Example 2 shown in Figure 7(a), applying this step yields the set of all groups S:

$$S: \{g4, g5, g7\} \; \{g2, g3, g6\} \; \{g1\}$$
$$\{g7\} \; \{g5, g6\} \; \{g1, g2\} \; \{g3, g4\}$$

We observe that some groups (e.g., $\{g1\}$ and $\{g7\}$) are subsets of other groups ($\{g1, g2\}$ and {g4, g5, g7}), and redundant groups are not allowed. To find such groups, the condition can be expressed as:

$$\forall X \in S \mid (Y \subseteq X \lor Y = X) \land (X \neq Y) \quad (5)$$

Or for every group X in S, Y is either a subset of X or equal to X. If the condition is met, Y is removed from S. As a result, S is given as:

$$S: \{g4, g5, g7\} \; \{g2, g3, g6\} \; \{g5, g6\} \; \{g1, g2\} \; \{g3, g4\}$$

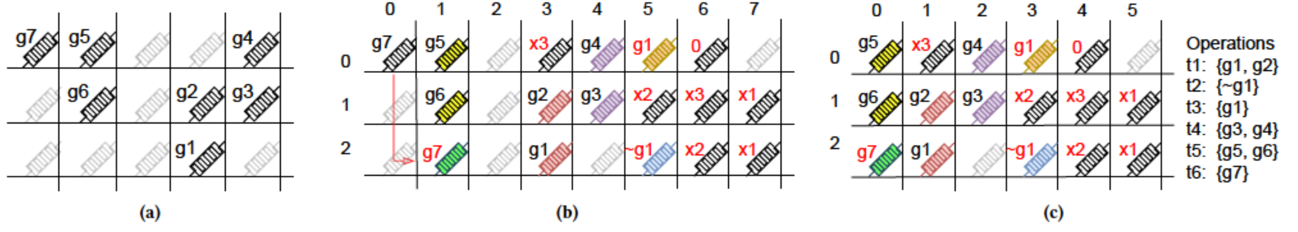

Figure 7: (a) Mapping obtained from genetic operators. (b) Mapping completion algorithm showcase. (c) Legal MAGIC mapping.

2.3.2 Resolve Data Dependency. If two gates, denoted as g_x and g_y, are connected sequentially within the same path, they cannot operate simultaneously due to data dependency. This implies that the output of one gate must be calculated and propagated before the output of the other gate can be determined. This condition can be expressed as:

$$\forall(g_x, g_y) \in G \mid (g_x \in path(g_y) \lor g_y \in path(g_x)) \tag{6}$$

The term $path(g_x)$ (or $path(g_y)$) denotes the route from any primary input to gate g_x (or g_y). Applying this step to the output calculated in step 1, further divides the groups as:

$$S : \{g4\}\ \{g5\}\ \{g7\}\ \{g6\}\ \{g2, g3\}\ \{g5, g6\}\ \{g1, g2\}\ \{g3, g4\}$$

After applying condition 6, some gates may form redundancy with others when disconnected from their group. Thus, condition 5 is reapplied.

$$S : \{g7\}\ \{g2, g3\}\ \{g5, g6\}\ \{g1, g2\}\ \{g3, g4\}$$

2.3.3 Find Directions. A direction is assigned to each gate g in G denoted as $D(g)$ as follows:

$$D(g) = \begin{cases} \text{EAST} & \text{if } fanin(g) : \text{EAST} \\ \text{WEST} & \text{if } fanin(g) : \text{WEST} \\ \text{NORTH} & \text{if } fanin(g) : \text{NORTH} \\ \text{SOUTH} & \text{if } fanin(g) : \text{SOUTH} \lor \\ & (fanin(g)[0] \lor fanin(g)[1]) : \text{NORTH} \\ \text{EAST} & \text{otherwise} \end{cases} \tag{7}$$

g is directed EAST if the fanins are located eastward relative to g's location, and similarly for every case. If one of the fanins of g is located southward but the other one is located northward relative to g, then a default direction of SOUTH is assigned. In all other cases, the default direction to assign is EAST. After determining gate directions, only gates with matching directions can form groups; otherwise, they are disconnected from the group. Note that condition 5 is also reapplied after this step to eliminate redundancies. Applying these conditions to the output of step 2 makes no change as the directions of all gates are computed EAST; hence, S remains unchanged:

$$S : \{g7\}\ \{g2, g3\}\ \{g5, g6\}\ \{g1, g2\}\ \{g3, g4\}$$

2.3.4 Resolve Conflicts. There may arise situations where a gate g belongs to multiple groups in S. However, it's necessary for each gate to belong to only one group. To identify such gates, the condition can be expressed as:

$$\forall(X, Y) \in S \mid (g \in X \cap Y) \land (X \neq Y) \tag{8}$$

To resolve conflicting gates, we establish criteria to determine which group it remains in and from which it is removed. The simplest criterion is to group it with the set that contains the most number of gates.

$$X = \arg\max_{X \in S} |X| : g \in X \tag{9}$$

If g exists in multiple groups with equal lengths, it is paired in a way that avoids disconnecting its group members from other groups. For instance, in set S of Example 2, $g2$ is part of $\{g1, g2\}$ and $\{g2, g3\}$, while $g3$ is part of $\{g2, g3\}$ and $\{g3, g4\}$. If we decide to group $g2$ with $g3$ in $\{g2, g3\}$ then two groups $\{g1, g2\}$ and $\{g3, g4\}$ would need to be let go. Thus, the preferred choice is to let go of $\{g2, g3\}$, making the other two groups feasible. Then, S is given as:

$$S : \{g7\}\ \{g5, g6\}\ \{g1, g2\}\ \{g3, g4\}$$

After this step, each group contains gates that are not shared with any other group.

2.3.5 Relocate Gates. Gate outputs that are not bound in a group are relocated on the crossbar to eliminate unnecessary operations. The condition can be expressed as:

$$\forall X \in S \mid (X \subseteq PO \land |X| = 1 \land aligned(fanin(g)[0], fanin(g)[1]) \land \neg aligned(g, fanin(g))) \tag{10}$$

Alternatively, for every group X in S such that it contains a single gate g, and X is either a subset of or equal to the set of POs denoted by PO, and the $fanins(g)$ are aligned on the crossbar, whether in rows or columns, but g is not aligned with its fanins. Applying condition 10 to Example 2, the gate $g7$ is identified as a potential candidate that can be relocated to align with its inputs $g5$ and $g6$ in column 1 as shown in Figure 7(b).

2.3.6 Map Copy Operations. If a gate g_y has gate g_x as fanin but the two are not aligned in either rows or columns. Then g_x is copied according to Algorithm 1 to align with g_y while considering the directional constraints. The condition can be expressed as:

$$\forall(g_x, g_y) \in G \mid (fanin(g_y) = g_x \land \neg aligned(g_x, g_y)) \tag{11}$$

Applying condition X to Example 2, $g1$ is identified as a candidate since it serves as a fanin to $g4$, is not aligned, and is bound in a group with $g2$. Therefore, its value must be copied from location $(2, 3)$ to row 0, aligning it with $g4$. This alignment should occur in the same row as per the directional constraints obtained for $g4$ in step 3. If Algorithm 1 fails to find a 2-hop path, then an empty row (column) is added to the crossbar and the algorithm is retried. Note that Algorithm 1 applies when row alignment is required; the same algorithm can be adapted and applied for column alignment.

Algorithm 1 Find_two_hop_path

```
Input: Crossbar M, starting coordinate start_coord, target row tgt_row
Output: 2-hop path from start_coord to tgt_row
 1: num_rows, num_cols ← Dimensions of M
 2: memo ← Empty dictionary
 3: function CAN_HOP(curr, prev, steps_left)
 4:     if Memoized result exists then
 5:         return it
 6:     if curr reaches target in steps_left steps then
 7:         return path
 8:     if No steps left then
 9:         return failure
10:     Extract row, col from curr
11:     for valid neighbors r, c do              ▷ Up/down or left/right moves
12:         Recursively call neighbor, updating memo on success
13:     Update memo and return failure
14: Call CAN_HOP(start_coord, None, 2)
15: if successful then
16:     return path
17: else
18:     return failure
```

2.3.7 Cleanup. In the final step, any rows or columns containing only free memristors are deleted. For instance, in Figure 7(b), columns 0 and 2 are removed, as depicted in Figure 7(c).

After completing the aforementioned steps, the mapping is prepared for evaluation. The number of utilized memristors and crossbar dimensions are counted, while the required clock cycles are calculated using the following formula:

$$\begin{aligned} Cycles = & \text{ No. of groups } + \text{ No. of loner gates} \\ & + (2 \times \text{No. of copy ops}) \end{aligned} \tag{12}$$

3 EXPERIMENTS

3.1 Experimental Setup

We have studied the performance of our algorithm on 8 benchmarks from IWLS-93, with plans to investigate more in a long-format article. The profiles of these benchmarks obtained from the Berkeley-ABC system formatted as **Name: I/O (NIG Nodes)** are listed below:

5xp1 :	7/10(131),	**misex1 :**	8/7(81),
b12 :	15/9(94),	**misex2 :**	25/18(159),
clip :	9/5(133),	**rd73 :**	7/3(157),
cordic :	23/2(93),	**inc :**	7/9(149)

The selection is influenced by the fact that the proposed algorithm is computationally intensive; hence, it is reasonably scalable. According to our preliminary testing, it should be able to easily optimize mapping of circuits up to 8- or 16-bit compressors and 8-bit multipliers within reasonable run times, making it ideal for the design of MAC units, convolution filters, and other frequently used arithmetic circuits in DNNs. In addition, with seeded initial conditions, it is anticipated to be *widely* scalable. Nonetheless, further investigation is still needed to establish the upper bounds on its scalability with greater accuracy and confidence. In our efforts to provide a thorough analysis, we observe that this benchmark size could be considered smaller than ideal. This limitation is primarily due to constraints on space and resources inherent to this article. Nevertheless, we have made considerable efforts to select representative benchmarks that provide meaningful insights into the proposed methodology.

The experiments are set up as follows. Evolution is implemented as a multi-objective optimization problem, focusing on solution time cycles and memristor count. Evolution halts after a predefined amount of time (12 hours in this case). We did not aid the convergence through seeding, rather, we used random initial conditions for the solutions where one instance per gate output in the NIG is randomly placed on the crossbar. This allows us to investigate the exploratory and convergence potential of the proposed method when starting with random solutions. With this setup, we conducted 5 independent runs on each benchmark, collected data on Pareto fronts for each run and tracked the best minimum value of each objective across all populations in each generation for convergence analysis. The application code was implemented to utilize multiple processes controlled by $n_process$ facilitating parallel evaluation of solution fitnesses across all populations. These experiments were conducted on a Linux machine with 64GB of memory and an AMD EPYC 64-core processor. The hyperparameter values for the genetic algorithm (GA) were adopted from standard literature:

$n_process$:	32,	$cxpb$:	0.7,
n_pops :	50,	$mutpb$:	0.3,
n_ind :	50,	mig_rate :	0.1,
$mig_interval$:	10		

3.2 Results and Discussion

The goal of the algorithm is to maximally minimize latency (Cycles) and memristor utilization on crossbar (#Mem). Pareto fronts obtained from five independent runs for each benchmark are depicted in Figure 8. The solutions are visualized as scatter plots, with each run represented by distinct colors. It is important to note that the objectives, Cycles and #Mem, are simultaneously minimized. While a reduction in one objective generally corresponds to a decrease in the other, this relationship is neither strictly inverse nor linear, as demonstrated in Example 1 above. Figure 8 illustrates the consistent performance achieved by the proposed approach. Minimal deviations are observed across all independent runs, indicating robustness in the generated Pareto fronts.

We compare our results with those presented in [7], which utilizes a Simulated Annealing-based approach. Our evaluation assesses the effectiveness of our problem formulation and island-style Genetic Algorithm (GA) against their single-population approach. Unlike our multi-objective island0-style GA approach yielding multiple results, their method yields a single result, with a cost function designed to handle one objective at a time. For the same benchmark, their result is plotted as a red triangle in Figure 8. The Pareto fronts from all five runs consistently dominate those of [7] across all benchmarks, highlighting the superior optimization capability of the proposed method. The figure axes are scaled to emphasize details, consistently positioning the competition's marker in the upper right corner.

We also monitored the solutions achieving the best (minimum) values for Cycles and #Mem objectives across all populations in each run. The convergence curves for all benchmarks are depicted

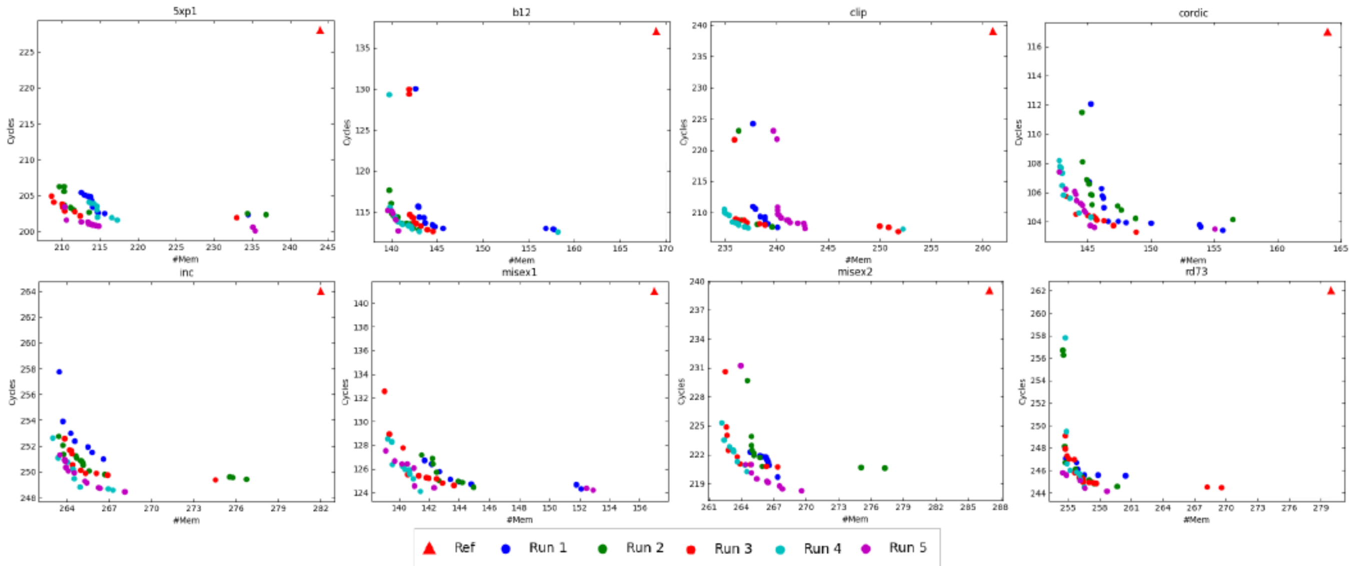

Figure 8: All Pareto efficient solutions from 5 test runs for each benchmark.

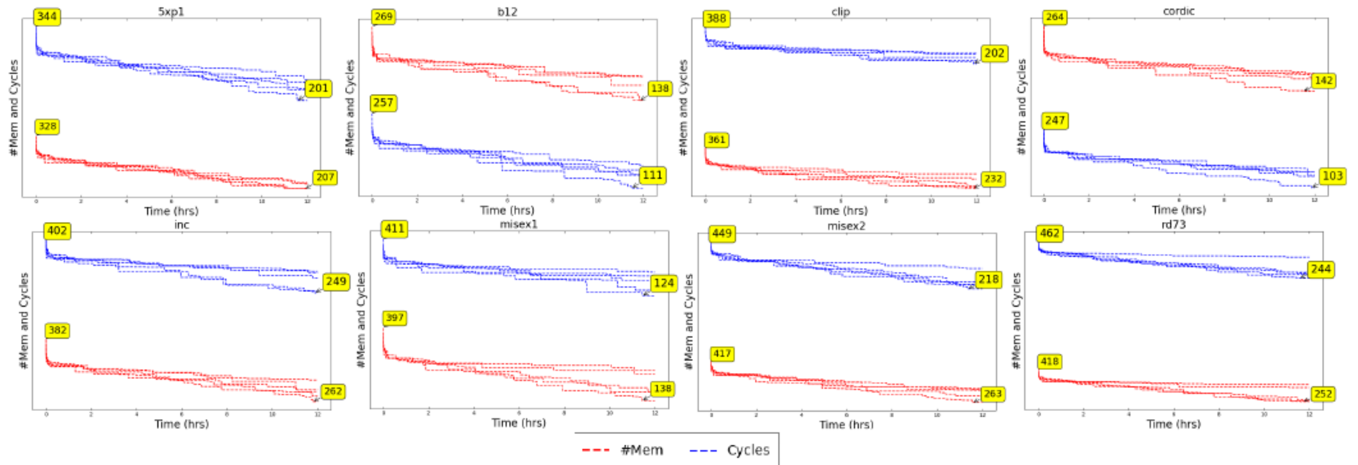

Figure 9: Convergence curves from 5 test run for each benchmark.

in Figure 9. Initial objective values and the minimum values near the end of each run are annotated in the figure. We observe significant improvements in both objectives typically occurring within the first hour of each run, as indicated by sharp declines in the early curves. Subsequently, optimization progresses steadily but less aggressively. The figure illustrates minimal deviations in convergence curves among independent runs across all benchmarks, demonstrating consistency in the algorithm's performance.

4 CONCLUSION

We have addressed the problem of in-memory mapping onto memristor crossbars. An automatic design method was proposed to handle multiple objectives and produce a set of non-dominated points. The proposed algorithm underwent evaluation using a subset of IWLS benchmarks. Experimental results demonstrate the algorithm's exploratory capability and consistent performance across multiple independent runs for each benchmark. Future endeavors include expanding evaluations to encompass larger circuits to accurately gauge the algorithm's scalability limits. Additionally, extended runs on arithmetic circuits are planned to compile a reusable macro database.

5 ACKNOWLEDGMENT

This work was supported in part by the Institute of Information and communications Technology Planning and Evaluation (IITP) grant funded by the Korea Government (MSIT) through Logic Synthesis for NVM-based PIM Computing Architecture under Grant 2022-0-00971. The EDA tool was supported by the IC Design Education Center (IDEC), Korea.

REFERENCES

[1] Rahul Gharpinde et al. 2018. A scalable in-memory logic synthesis approach using memristor crossbar. *IEEE Trans. on VLSI Systems* 26, 2 (Feb. 2018), 355–366.
[2] Song Han et al. 2016. EIE: Efficient inference engine on compressed deep neural network. *ACM SIGARCH Computer Architecture News* 44, 3 (June 2016), 243–254.
[3] Rotem Ben Hur et al. 2017. SIMPLE MAGIC: Synthesis and in-memory mapping of logic execution for memristor-aided logic. In *Proc. Int. Conf. on Computer-Aided Design*. 225–232.
[4] Shahar Kvatinsky et al. 2011. Memristor-based IMPLY logic design procedure. In *Proc. Int. Conf. on Computer Design*. 142–147.
[5] Shahar Kvatinsky et al. 2014. MAGIC—memristor-aided logic. *IEEE Trans. on Circuits and Systems II* 61, 11 (Nov. 2014), 895–899.
[6] Lauren Leffer. 2023. The AI boom could use a shocking amount of electricity. https://www.scientificamerican.com/article/the-ai-boom-could-use-a-shocking-amount-of-electricity/. [Online; accessed 03-April-2024].
[7] Phrangboklang L Thangkhiew and Kamalika Datta. 2018. Scalable in-memory mapping of Boolean functions in memristive crossbar array using simulated annealing. *Journal of Systems Architecture* 89 (Sept. 2018), 49–59.
[8] Zhenhua Zhu et al. 2019. A general logic synthesis framework for memristor-based logic design. In *Proc. Int. Conf. on Computer-Aided Design*. 1–8.
[9] Alwin Zulehner et al. 2019. A staircase structure for scalable and efficient synthesis of memristor-aided logic. In *Proc. Asia and South Pacific Design Automation Conf.* 237–242.