EasyChair Preprint

№ 13769

Integrated Netlist Synthesis and In-Memory Mapping for Memristor-Aided Logic

Seunggyu Lee, Wonjae Lee and Youngsoo Shin

July 2, 2024

Integrated Netlist Synthesis and In-Memory Mapping for Memristor-Aided Logic

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

Wonjae Lee
School of EE, KAIST
Daejeon, Korea
wonjaelee@kaist.ac.kr

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

ABSTRACT

Memristive memory (memristor) enables logic operations within the memory array, where memristors in the same row or column serve as a logic gate. Logic functions are implemented in the memory through netlist synthesis and in-memory mapping, which assigns each gate operation to specific memristors. The goal is to minimize latency, which represents the number of clock cycles required to complete the operations. While multiple gate operations can be executed in the same clock cycle, additional cycles may be needed for copy operations to align the gate operations. Therefore, assigning each operation to a clock cycle is a challenge. Furthermore, the results of in-memory mapping vary depending on the input netlist. To further reduce latency, an integrated approach is necessary to provide an optimal netlist. We propose two approaches: (1) graph coloring-based in-memory mapping, where the gates are colored to assign sets of gates that operate simultaneously, and (2) integration with mapping-aware netlist synthesis, which iteratively revises the input netlist based on latency evaluation; an incremental method is employed to accelerate the process. Experiments demonstrate that the coloring-based in-memory mapping reduces latency by 17% compared to the state-of-the-art method. The integrated approach achieves an additional 15% reduction in latency.

CCS CONCEPTS

• **Hardware** → **Emerging technologies**; **Logic synthesis**.

KEYWORDS

Memristor-aided logic, logic synthesis, in-memory mapping, graph coloring.

ACM Reference Format:
Seunggyu Lee, Wonjae Lee, and Youngsoo Shin. 2024. Integrated Netlist Synthesis and In-Memory Mapping for Memristor-Aided Logic. In *Great Lakes Symposium on VLSI 2024 (GLSVLSI '24), June 12–14, 2024, Clearwater, FL, USA.* ACM, New York, NY, USA, 6 pages. https://doi.org/10.1145/3649476.3658758

1 INTRODUCTION

Conventional logic operation is performed within a processing unit (i.e., CPU or GPU), and processed data is transferred into the memory for storage. The data transfer between the processor and the memory causes a significant overhead in both energy and performance; for instance, a DRAM access consumes three orders of magnitude more energy than a 32-bit add operation [5]. Processing-in-memory (PIM) allows logic operations to be conducted within the memory itself, so it significantly reduces the data transfer. Notably, PIM achieves 110 times greater energy efficiency [12] and 23 times higher performance [4].

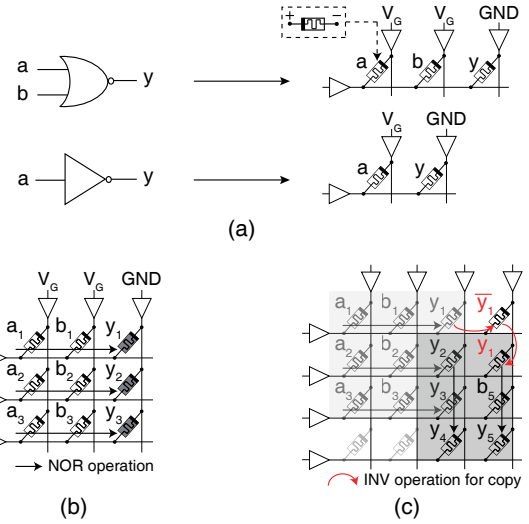

Figure 1: (a) NOR and INV gate diagrams and their realization using memristors, (b) a parallel operation for three NOR gates, and (c) a copy operation for alignment of parallel operations.

There have been prior works to implement processing units within memory cells, such as SRAM [1] and DRAM [10]. However, they still require data transfer between the processing and the storage. In this paper, we use a non-volatile resistive memory called memristor [3], which is the two-terminal device with two resistance values. The resistance is determined by the voltage across the memristor, and then its resistance state is preserved. Low and high resistance states represent logic 1 and 0, respectively. So, logic gates are implemented solely by memristors, called memristor-aided logic [7].

In memristor-aided logic, a logic function can be realized as a sequence of NOR and INV operations. All logic gates in the corresponding netlist are mapped onto memristive memory array. This process, which is known as in-memory mapping, specifies which memristors are used to perform the gate operation (i.e., NOR or INV) at a given clock cycle. As shown in Figure 1(a), NOR and INV gates are implemented by one or more input memristors and a single output memristor in a row (or column) of memristive memory array. The gate operation is executed by applying a gate voltage (V_G) to each input memristor and grounding the output memristor

GLSVLSI '24, June 12–14, 2024, Clearwater, FL, USA
2024. ACM ISBN 979-8-4007-0605-9/24/06.
https://doi.org/10.1145/3649476.3658758

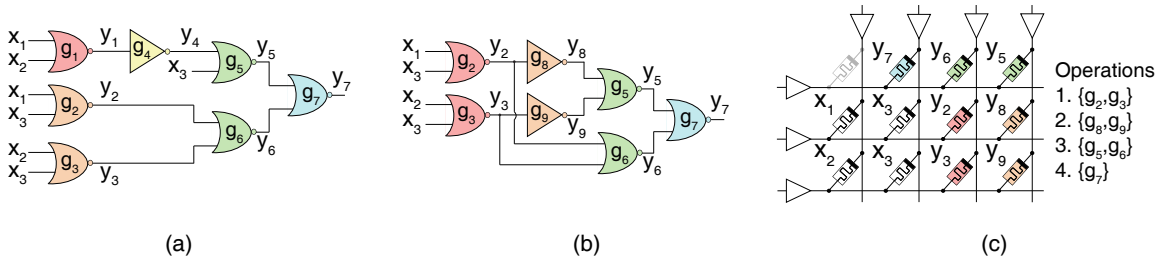

Figure 2: The NOR and INV netlist (a) colored through graph coloring formulation, (b) transformed into a netlist with minimum latency using the proposed netlist synthesis, and (c) mapped onto memristive memory array by formulating CP.

simultaneously, in which the gate inputs a, b and the gate output y correspond to resistance states of the input memristors and the output memristor, respectively. NOR or INV operations can be conducted in parallel by aligning the respective inputs and outputs of multiple gates on the same column (or row), as illustrated in Figure 1(b). It should also be noted that additional copy operations may be required due to the alignment for parallel operations in the next clock cycle (see y_1 in Figure 1(c)); the copy uses two INV operations to move the resistance state of one memristor to another.

The objective of in-memory mapping is to minimize the total number of clock cycles required for completing all gate operations, called latency. Staircase-structure mapping [15] aims to maximize parallel operations, but it also causes copy operations to align the inputs and output for multiple gates at each level of the netlist. In [2], latency optimization problem is formulated through integer linear programming (ILP). It is very time-consuming for large circuits because the optimizer searches all possible operations by mapping gates to various locations. To reduce its runtime, multiple ILPs are used [14] by partitioning the netlist. However, a large number of copy operations are required for connecting the partitions. In addition, none of these methods consider latency during netlist synthesis; instead, they just use the conventional netlist synthesis which aims to minimize the area (i.e., the number of gates in the netlist).

In this paper, we address a mapping method using graph coloring, and its integration with mapping-aware netlist synthesis. First, coloring-based in-memory mapping is introduced to maximize parallel operations without copies. Parallel operations are assigned through graph coloring formulation such that gates with the same color operate in parallel. For the colored netlist, in-memory mapping is performed by formulating constraint programming (CP). Second, in-memory mapping is integrated with the proposed netlist synthesis, which utilizes a rewriting algorithm and an incremental graph coloring method. The rewriting algorithm iteratively performs local replacements in the netlist to minimize latency, and the incremental graph coloring accelerates the iterative latency evaluations by reusing an existing coloring result.

Our main contributions are summarized as follows.

- In-memory mapping based on graph coloring, which maximally assigns parallel operations without copy overhead.
- Integrated approach, which synthesizes the netlist in correlation with in-memory mapping to further reduce latency through local replacements and quick latency evaluation.

The remainder of this paper is organized as follows. The details of the integrated netlist synthesis and in-memory mapping are presented in Section 2. The effectiveness of the proposed method is assessed in terms of latency in Section 3. Conclusions are drawn in Section 4.

2 PROPOSED METHODS

Figure 2 illustrates an overview of our three approaches, where x and y_i denote the primary input and the gate g_i output, respectively. First, a NOR and INV netlist is colored by graph coloring formulation. In this process, sets of parallel operations are assigned in such a way that gates capable of parallel operation share the same color (Figure 2(a)). Since each parallel operation takes one clock cycle, the number of colors directly represents latency. Second, the proposed netlist synthesis transforms a netlist into logically equivalent one with the minimum number of colors (Figure 2(b)). Third, constraint programming (CP) is formulated to perform in-memory mapping for a colored netlist. As a result, we obtain an operation sequence where gates with the same color operate simultaneously in each clock cycle, and a mapping result in which the inputs and outputs of gates are aligned within the memristive memory array; Figure 2(c) shows the mapping result of Figure 2(b). The proposed coloring-based mapping utilizes the first and third approaches, while the integrated method applies all three approaches.

2.1 Parallel Operation Assignment Using Graph Coloring

A graph coloring problem is formulated based on four rules that reflect an efficient in-memory mapping approach, which maximizes parallel operations without requiring copies. To solve the formulation, we employ a graph coloring algorithm [8], which yields the best solution compared to other heuristic algorithms. Then, we derive a colored netlist which consists of gates with corresponding colors. Note that this graph coloring result is utilized in the proposed netlist synthesis (Section 2.2), which finds a netlist with the minimum number of colors, and in the CP-based in-memory mapping (Section 2.3), where multiple gates sharing the same color are aligned.

Figure 3 shows an example of graph coloring formulation. NOR and INV netlist is modeled as an undirected graph, where a vertex v_i corresponds to a gate g_i and an edge e_{ij} connects two vertices v_i and v_j when they satisfy any of the three conditions.

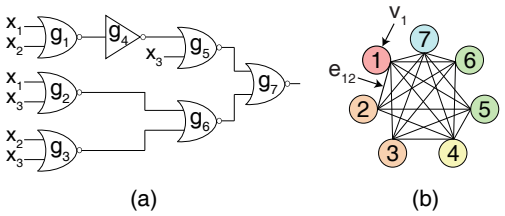

Figure 3: (a) An example NOR and INV netlist and (b) the result of its graph coloring formulation.

2.1.1 Signal Dependency. Signal path is a connection of gates where signals propagate from primary inputs to primary outputs. If two gates are part of the same signal path, the output of one gate propagates through intermediate gates and then serves as the input of the other gate. As a result, they cannot operate in parallel due to their signal dependency. For instance, as shown in Figure 3(a), the output of gate g_1 is provided to gate g_5 through gate g_4, so this constructs the signal dependency between gates g_1 and g_5. In the same manner, all gates in the signal path $\{g_1, g_4, g_5, g_7\}$ cannot be conducted in parallel. Therefore, an edge is added between each pair of vertices v_1, v_4, v_5, and v_7.

2.1.2 Gate Type. NOR and INV gates operate in separate clock cycles because they use different numbers of input memristors. Thus, we add an edge between vertices with different gate types. As shown in Figure 3(b), the vertex v_4 corresponding to the INV gate g_4 is connected to all vertices that are associated with NOR gates. In addition, we consider the gate connected to the fanout, called fanout gate. It is necessary to perform continuous parallel operations instead of maximizing parallel operations at specific clock cycles; otherwise, there would be no more parallel operations in the next cycles. Accordingly, gates with fanouts connected to different NOR gates should not operate in parallel. If the fanout gates are different and at least one of them is a NOR gate, an edge connects between the corresponding vertices. As illustrated in Figure 3(b), an edge e_{12} is generated because the fanouts of the associated gates g_1 and g_2 are connected to INV gate g_4 and NOR gate g_6, respectively. Note that if all fanout gates are of type INV, no edges are constructed. This is because INV gates use only one input memristor, which does not affect alignment.

2.1.3 Number of Fanouts. An output of a gate operation can be stored in multiple memristors, which is achieved by grounding the output memristors at a clock cycle. This allows for providing inputs to the gates connected to fanouts in a single cycle, and then enables those gates to operate in parallel. If the number of fanouts differs between two gates, they use a different number of output memristors, which results in them executing at separate clock cycles. Thus, we create edges between gates with different numbers of fanouts. Also, the gate connected to the fanin, called fanin gate, is taken into account. The difference in the number of fanouts for fanin gates leads to those gates operating on different clock cycles. It affects the mapping of the next gates connected to the fanouts of the fanin gates. To reflect this condition for fanin gate, we compare sets that consist of the number of fanouts. If the sets for the fanin gates are different, we add an edge between the vertices corresponding to the two gates.

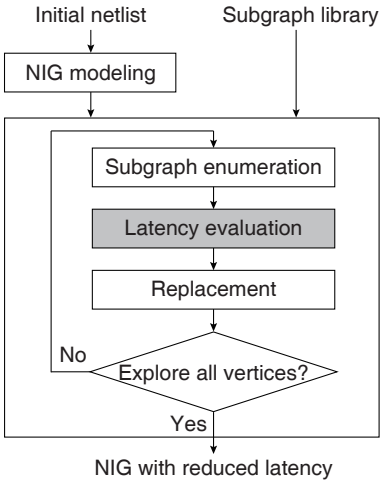

Figure 4: Overall flow of NIG rewriting.

2.2 Mapping-Aware Netlist Synthesis

A logic function is represented as a NOR and INV netlist. To decrease the latency of the resulting netlist, we propose the NIG rewriting method, which transforms the netlist into another logically equivalent one with minimum latency. The latency is presented as the number of colors derived from our graph coloring formulation, eliminating the need to calculate latency through in-memory mapping; the runtime for executing mapping once is much larger than the time taken for a single graph coloring formulation.

Even though performing the graph coloring algorithm [8] once is fast, the total runtime for latency evaluation largely increases with the number of gates in the netlist. It is because as the netlist size is larger, the number of enumerated subgraphs increases. This results in a significant number of graph coloring formulations. In addition, the coloring algorithm exhibits $O(n^2)$ time in the n-vertex graph, where n denotes the number of gates in a netlist. To significantly reduce the runtime, we propose an incremental graph coloring method. It revises an existing coloring result, instead of performing entirely new graph coloring formulation.

2.2.1 NIG Rewriting. Figure 4 shows the overall flow of the proposed rewriting method. We model an input netlist as NOR-INV graph (NIG), where vertices correspond to NOR gates while edges represent gate connections. INV gates are denoted by black markers on the edges. To explore the logically equivalent netlists, a library is created from pre-computed library [9] which stores logically equivalent AIG subgraphs for 4-variable functions; NIG is obtained by converting each AND gate of AIG into NOR and INV gates, and then NIG subgraph library is derived.

For each vertex of NIG, we perform subgraph enumeration, which searches all logically equivalent subgraphs in the NIG subgraph library; input NIG can be transformed into an equivalent NIG by using those subgraphs. Each of the equivalent NIG is evaluated in latency through the proposed incremental graph coloring

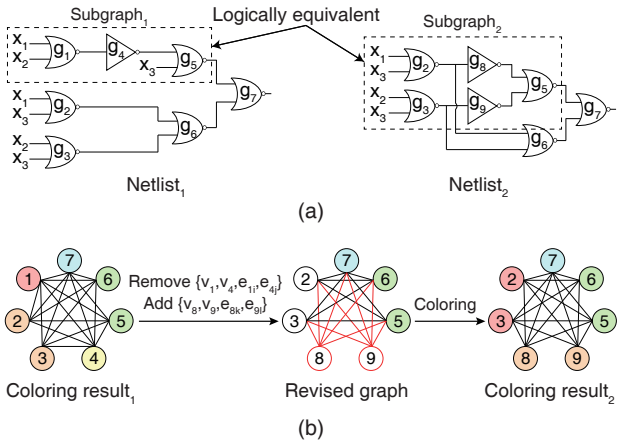

Figure 5: (a) An example of two logically equivalent netlists and (b) incremental graph coloring.

method. Finally, among the equivalent subgraphs, the subgraph of input NIG is replaced with the optimal subgraph that minimizes the latency. The same process sequentially repeats for all vertices in topological order.

2.2.2 Incremental Graph Coloring. Figure 5 shows an example of incremental graph coloring method, where subgraph_1 in the netlist_1 is logically equivalent with subgraph_2 in the netlist_2 (see dotted lines in Figure 5(a)). A coloring result_1 for netlist_1 is obtained using an algorithm introduced in [8]. Then, we revise the coloring result_1 to formulate the graph coloring problem for netlist_2. The subgraph_1 vertices v_1 and v_4 that are not in subgraph_2 are removed with those edges e_{1i} and e_{4j}, while subgraph_2 vertices v_8 and v_9, which are not present in netlist_1, are added with their corresponding edges e_{8k} and e_{9l} (see red lines in Figure 5(b)). Finally, except for the explored gate g_5, all subgraph_2 vertices v_2, v_3, v_8, and v_9 are colored by the graph coloring algorithm, resulting in the coloring result_2 for netlist_2.

The incremental method confines vertex exploration within the subgraph. This ensures that regardless of the netlist size, coloring is only applied to vertices of the subgraph. In addition, the remaining vertices outside the subgraph have pre-determined colors. Thus, the complexity is significantly reduced to $O(m^2)$, where m denotes the number of gates in the subgraph. Reusing the existing coloring result may slightly increase the number of colors used, but the impact is small. This is because, despite the subgraph replacement, the relationships between vertices outside the subgraph are preserved.

2.3 In-Memory Mapping Using CP

In-memory mapping is performed by formulating constraint programming (CP) for a colored netlist. In this formulation, the locations of both the input and output memristors for each gate are determined while satisfying the alignment constraints. Operation sequence is derived from colored gate sets, in which gates sharing the same color execute parallel operations, while gates with different colors operate in different clock cycles. The formulated CP can be quickly solved because its goal is to find a solution that satisfies the alignment constraints, unlike previous works [2, 14] that solve

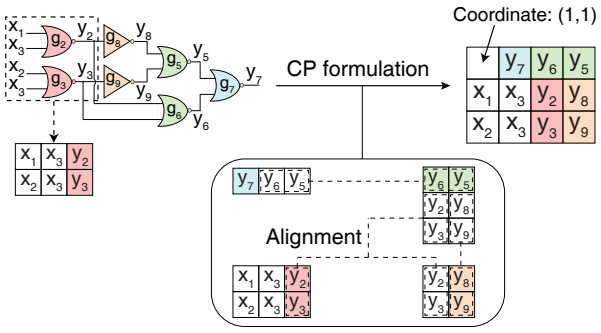

Figure 6: In-memory mapping through CP formulation, which perform alignments between parallel operation sets.

the latency optimization problem. In addition, parallel operation sets, which are obtained from the colored gates, are provided to CP. Thus, alignments are performed based on the sets, instead of individual gates, as shown in Figure 6.

2.3.1 CP Formulation. We utilize row and column indices as coordinates for both the input and output memristors. The coordinate of NOR gate g_i is defined as:

$$\left\{(r_{in1_{g_i}}, c_{in1_{g_i}}), (r_{in2_{g_i}}, c_{in2_{g_i}}), (r_{out_{g_i}}, c_{out_{g_i}})\right\}, \tag{1}$$

where the gate inputs $in1_{g_i}$, $in2_{g_i}$, and the gate output out_{g_i} are mapped to $(r_{in1_{g_i}}, c_{in1_{g_i}})$, $(r_{in2_{g_i}}, c_{in2_{g_i}})$, and $(r_{out_{g_i}}, c_{out_{g_i}})$, respectively, with row indices $r_{in1_{g_i}}, r_{in2_{g_i}}, r_{out_{g_i}} \in \mathbb{N}$ and column indices $c_{in1_{g_i}}, c_{in2_{g_i}}, c_{out_{g_i}} \in \mathbb{N}$. For INV gate g_i, only one input is used for its coordinate, i.e., $\left\{(r_{in1_{g_i}}, c_{in1_{g_i}}), (r_{out_{g_i}}, c_{out_{g_i}})\right\}$. For instance, in Figure 6, the coordinates of NOR gate g_2 and INV gate g_8 are $\{(2,1),(2,2),(2,3)\}$ and $\{(2,3),(2,4)\}$, respectively.

Each gate in a netlist operates in a row or a column. For a gate g_i, the inputs $in1_{g_i}$, $in2_{g_i}$ and the output out_{g_i} are arranged either in the same row and different columns or in the same column and different rows, which is given by:

$$\left\{\left(r_{in1_{g_i}} = r_{in2_{g_i}} = r_{out_{g_i}}\right) \,\&\, \left(c_{in1_{g_i}} \neq c_{in2_{g_i}} \neq c_{out_{g_i}}\right)\right\} \tag{2}$$
$$\| \left\{\left(c_{in1_{g_i}} = c_{in2_{g_i}} = c_{out_{g_i}}\right) \,\&\, \left(r_{in1_{g_i}} \neq r_{in2_{g_i}} \neq r_{out_{g_i}}\right)\right\}.$$

Since gates g_i and g_j with the same color operate in parallel, they should be aligned in columns or rows. This is represented by:

$$\left[\left\{\left(c_{in1_{g_i}} = c_{in1_{g_j}}\right) \,\&\, \left(c_{in2_{g_i}} = c_{in2_{g_j}}\right) \,\&\, \left(c_{out_{g_i}} = c_{out_{g_j}}\right)\right\}\right. \tag{3}$$
$$\| \left.\left\{\left(c_{in1_{g_i}} = c_{in2_{g_j}}\right) \,\&\, \left(c_{in2_{g_i}} = c_{in1_{g_j}}\right) \,\&\, \left(c_{out_{g_i}} = c_{out_{g_j}}\right)\right\}\right]$$
$$\| \left[\left\{\left(r_{in1_{g_i}} = r_{in1_{g_j}}\right) \,\&\, \left(r_{in2_{g_i}} = r_{in2_{g_j}}\right) \,\&\, \left(r_{out_{g_i}} = r_{out_{g_j}}\right)\right\}\right.$$
$$\| \left.\left\{\left(r_{in1_{g_i}} = r_{in2_{g_j}}\right) \,\&\, \left(r_{in2_{g_i}} = r_{in1_{g_j}}\right) \,\&\, \left(r_{out_{g_i}} = r_{out_{g_j}}\right)\right\}\right],$$

where the gates g_i and g_j should simultaneously satisfy Equation (2). If a gate output out_{g_i} serves as another gate input in_{g_j} (i.e., $in1_{g_j}$ or $in2_{g_j}$), out_{g_i} and in_{g_j} are mapped to the same memristor. The

connectivity between gates g_i and g_j is given as:

$$\left(r_{out_{g_i}} = r_{in_{g_j}}\right) \,\&\, \left(c_{out_{g_i}} = c_{in_{g_j}}\right). \tag{4}$$

To preserve resistance states of output memristors, the outputs out_{g_i} and out_{g_j} of two different gates g_i and g_j should not overlap; thus, they are mapped to different memristors, which is represented as:

$$\left(r_{out_{g_i}} \neq r_{out_{g_j}}\right) \,||\, \left(c_{out_{g_i}} \neq c_{out_{g_j}}\right). \tag{5}$$

3 EXPERIMENTAL RESULTS

A set of test circuits from ISCAS85 benchmarks [6] are taken for the experiments, which are listed in Table 1 in order of circuit complexity. Conventional netlist synthesis is conducted using a standard logic synthesis tool (ABC) [11], and the number of gates in the resulting netlist is presented in column 2 of Table 1. Our netlist synthesis and in-memory mapping are implemented in C++. The number of colors in the graph coloring formulation is calculated by a coloring algorithm [8]. The solutions of ILP and CP are obtained using a commercial solver [13].

3.1 In-Memory Mapping

In Table 1, we assess our in-memory mapping in terms of latency. Our proposed method (coloring) is compared with two previous approaches: staircase method [15] and partitioning-based method [14]. Each test circuit is synthesized into a NOR and INV netlist through conventional netlist synthesis. The netlist is mapped by our coloring-based method, and the resulting latency is presented in column 5. Previous methods are also applied to the same netlist, and the latency differences from our approach are shown in columns 3-4 (see parentheses). For the partitioning-based approach, we vary partition sizes and select the best result within the runtime limit of 24 hours; this ensures tractability because solving ILP for large partitions causes significant runtime overhead.

Our method consistently outperforms the previous approaches, demonstrating reliable improvements across all test circuits. Specifically, we achieve the minimum latency compared to both the staircase and partitioning-based methods, which increase the average latency by 31% and 17%, respectively. This is because our graph coloring formulation maximizes parallel operations without copy overhead. It is important to note that the proposed method shows greater latency reduction, particularly for larger circuits. This comes from the fact that copy operations in previous methods increase with the circuit size due to the larger complexity of the mapping process. In the staircase method, gates from each level are alternatively mapped in rows and columns. Then, inputs for the next level are provided through simple alignments using copy operations. Similarly, the partitioning-based approach formulates ILPs individually for each partition. Even though the resulting latency in a partition is minimized, a large number of copy operations are required to align between multiple partitions. On the other hand, our solution is not affected by such copy overhead.

We further evaluate our method in terms of efficiency by comparing its runtime with that of the partitioning-based approach. We gradually increase the partition size for the previous approach until it achieves a similar latency to ours, without setting any runtime limit. As a result, in the case of the smallest test circuit (c432), the partitioning-based method exhibits the same latency of 181; but it takes over a week, while our mapping is completed within 5 minutes. Notably, the maximum runtime from our method among all test circuits is only 50 minutes for c6288.

Table 1: Comparison of various in-memory mapping methods in latency

Circuits	#Gates	Latency		
		Staircase [15]	Partitioning [14]	Coloring (Proposed)
c432	251	225 (+24%)	204 (+13%)	181
c880	521	427 (+26%)	382 (+13%)	339
c1908	583	517 (+26%)	465 (+14%)	409
c499	606	242 (+27%)	219 (+15%)	190
c1355	606	236 (+27%)	216 (+16%)	186
c2670	1060	551 (+30%)	493 (+16%)	425
c3540	1422	1435 (+33%)	1270 (+18%)	1079
c5315	1989	1361 (+35%)	1196 (+18%)	1011
c7552	2345	2182 (+37%)	1913 (+20%)	1597
c6288	2718	3751 (+40%)	3491 (+31%)	2672
Average		+31%	+17%	

3.2 Integrated Method

The goal of the proposed integrated method is to achieve further reduction in latency by obtaining an optimal netlist based on the resulting latency of in-memory mapping. In Table 2, we assess our integrated method in terms of latency compared to our coloring-based mapping, where the input netlist is obtained from conventional netlist synthesis (see column 5 of Table 1). As listed in column 4, the latency difference compared to our coloring-based mapping is shown in parentheses, and the proposed integrated approach exhibits an additional 15% reduction in latency, on average. We observe that the latency difference increases with the circuit size. This is because in-memory mapping becomes more complex as the circuit is larger; the alignments between more gates are considered. Thus, to minimize the resulting latency of in-memory mapping, netlist synthesis should be performed in correlation with the mapping. But, conventional netlist synthesis simply reduces the area of the netlist, so it results in a netlist which requires more copy operations.

To assess the effect of the proposed incremental graph coloring, we create another integrated method as a baseline, where latency evaluation in NIG rewriting is performed using a non-incremental graph coloring method. The integrated method based on the non-incremental coloring significantly reduces latency as listed in column 2, but the results are shown only for 6 small circuits due to computational complexity, where the runtime limit is set to 24 hours. On the other hand, our incremental coloring-based method achieves latency reductions for all test circuits. As shown in columns 3 and 5, the runtime gain increases as the circuit is larger. This is because, when a circuit is modeled by NIG, both the number of vertices to explore and the number of subgraphs enumerated from those vertices increase in proportion to the circuit size. Accordingly, more graph coloring formulations for latency evaluation are required. Also, the runtime gain comes from the fact that the time taken for a single execution of the graph coloring algorithm [8] increases

Table 2: Comparison of two integrated methods in latency and runtime

Circuits	Non-incremental coloring		Incremental coloring	
	Latency	Runtime	Latency	Runtime
c432	158 (-13%)	28m	163 (-10%)	5m
c880	294 (-13%)	159m	304 (-10%)	13m
c1908	350 (-14%)	524m	363 (-11%)	15m
c499	162 (-15%)	605m	168 (-12%)	16m
c1355	159 (-15%)	595m	165 (-11%)	16m
c2670	352 (-17%)	1031m	366 (-14%)	19m
c3540	-	-	890 (-18%)	30m
c5315	-	-	804 (-20%)	36m
c7552	-	-	1264 (-21%)	39m
c6288	-	-	2094 (-22%)	52m
Average			-15%	

according to the time complexity (see Section 2.2). Even though the proposed integrated method achieves about 3% less reduction in latency compared to the non-incremental approach, this compromise is acceptable considering the significant runtime gain, e.g., 54 times for c2670. In addition, the runtime overhead resulting from integrating the proposed netlist synthesis is negligible; it's only about 3% for the largest test circuit (c6288). This indicates that our integration is very effective in reducing latency.

The resulting latency of in-memory mapping can vary significantly depending on the synthesized netlist. Figure 7 illustrates the changes in latency according to the number of replacements conducted during three netlist synthesis methods: conventional approach [9], NIG rewriting with non-incremental graph coloring method, and the proposed approach using incremental graph coloring. A test circuit (c432) serves as input for each netlist synthesis method. Conventional rewriting and the other two NIG rewriting methods perform replacements until the area and latency converge, respectively. We observe that the latency from the conventional netlist synthesis does not continue to decrease; in fact, latency even increases at specific replacements. In contrast, the other two methods consistently reduce latency with each replacement. This shows the importance of considering in-memory mapping during netlist synthesis.

4 CONCLUSION

We have addressed a coloring-based in-memory mapping, and its integration with mapping-aware netlist synthesis. Graph coloring formulation is performed to maximally assign parallel operations without copy overhead. The gates capable of parallel operation are assigned with the same color. In-memory mapping for the colored netlist is then performed by formulating constraint programming (CP). Our netlist synthesis employs a rewriting algorithm to obtain the netlist with minimum latency. The input netlist is iteratively transformed based on latency evaluation through an incremental graph coloring method. Experimental results show that the proposed coloring-based mapping achieves 17% reduction in latency compared to the state-of-the-art method. The integrated approach further reduces 15% of latency from our coloring-based method.

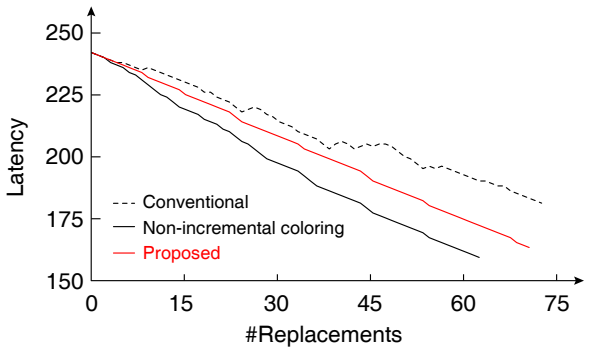

Figure 7: Latency per replacement conducted by conventional rewriting [9], NIG rewriting with non-incremental graph coloring method, and the proposed rewriting.

ACKNOWLEDGEMENTS

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] Shaizeen Aga et al. 2017. Compute caches. In *Proc. Int. Symp. on High Performance Computer Architecture*. 481–492.
[2] 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.
[3] Leon O. Chua. 1971. Memristor-the missing circuit element. *IEEE Trans. on Circuit Theory* 18, 5 (Sept. 1971), 507–519.
[4] Juan Gómez-Luna et al. 2022. Benchmarking a new paradigm: Experimental analysis and characterization of a real processing-in-memory system. *IEEE Access* 10 (May 2022), 52565–52608.
[5] Song Han et al. 2016. EIE: Efficient inference engine on compressed deep neural network. In *Proc. Int. Symp. on Computer Architecture*. 243–254.
[6] Mark C Hansen, Hakan Yalcin, and John P Hayes. 1999. Unveiling the ISCAS-85 benchmarks: A case study in reverse engineering. *IEEE Design & Test of Computers* 16, 3 (Sept. 1999), 72–80.
[7] Shahar Kvatinsky et al. 2014. MAGIC—memristor-aided logic. *IEEE Trans. on Circuits and Systems II* 61, 11 (Nov. 2014), 895–899.
[8] Frank Thomson Leighton. 1979. A graph coloring algorithm for large scheduling problems. *J. Res. Nat. Bur. Standards* 84, 6 (Dec. 1979), 489–506.
[9] Alan Mishchenko, Satrajit Chatterjee, and Robert Brayton. 2006. DAG-aware AIG rewriting: A fresh look at combinational logic synthesis. In *Proc. Design Automation Conf.* 532–535.
[10] Vivek Seshadri et al. 2017. Ambit: In-memory accelerator for bulk bitwise operations using commodity DRAM technology. In *Proc. Int. Symp. on Microarchitecture*. 273–287.
[11] Berkeley Logic Synthesis and Verification Group. 2012. ABC: A system for sequential synthesis and verification. http://www.eecs.berkeley.edu/~alanmi/abc/
[12] Peng Yao et al. 2020. Fully hardware-implemented memristor convolutional neural network. *Nature* 577, 7792 (Jan. 2020), 641–646.
[13] Z3Prover. 2024. The z3 theorem prover. https://github.com/Z3Prover/z3/
[14] 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.
[15] 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.