X-MAGIC: Enhancing PIM using Input Overwriting Capabilities

Natan Peled, Rotem Ben-Hur, Ronny Ronen, and Shahar Kvatinsky
Andrew and Erna Viterbi Faculty of Electrical Engineering
Technion - Israel Institute of Technology
Haifa, Israel 3200003
{natanpeled, rotembenhur}@campus.technion.ac.il, ronny.ronen@ef.technion.ac.il, shahar@ee.technion.ac.il

***Abstract*—Processing-in-memory (PIM) using memristive technologies is an attractive solution for the memory wall problem. PIM can improve the performance and energy efficiency of computing systems by reducing the data transfer between the memory and the processor. Memristor Aided loGIC (MAGIC) is a popular memristive PIM technique that can perform any combinational logic as a sequence of atomic NOR/NOT operations. These NOR/NOT operations rely on initializing their output cell prior to computation. In this paper, we explore input overwriting: the use of the MAGIC gate output cell as an additional input without initializing it. We extend MAGIC and introduce X-MAGIC (eXtended MAGIC) which uses input overwriting, and demonstrate it by two gates, $A \cdot (\overline{B+C})$ and $A \cdot \overline{B}$, where A is an overwritten input. We show that input overwriting improves functionality, performance, and effective lifetime of the system.**

Due to algorithmic difficulties, available PIM synthesis tools do not support input overwriting. We address these difficulties by modifying an existing synthesis tool for MAGIC (SIMPLER), and presenting several general principles and methods for supporting input overwriting. We examine two configurations of the modified synthesis tool using X-MAGIC gates, differing in their performance/area trade-off. Both configurations achieve a geomean improvement of over 16.5% in performance, and over 20% in effective lifetime compared to standard MAGIC.

***Index Terms*—memristor, logic synthesis, PIM, memory**

I. Introduction

In recent years, data transfer between the computation unit and the memory has become the main bottleneck in computer architecture [1]. One possible approach to alleviate this bottleneck, so to improve performance and reduce energy consumption, is the decades-old concept of processing-in-memory (PIM). Recent advances in memory technologies, and specifically, the emergence of new electronic devices, have brought PIM to the center-stage of contemporary computer systems research.

One example of such an emerging electronic device is the memristor [2]. In memristors, the resistance can be varied by applying current or voltage across the device. Hence, memristors can modulate data into resistance. Memristors can be used to build standard memories [3], and can even implement logic gates [4]–[6]. Furthermore, memristors can combine both capabilities [7], [8], *i.e.*, they can integrate data storage and computation to perform PIM.

Memristor aided logic (MAGIC) [4] is a technique which uses memristors to perform logic, and is compatible with memristive crossbar memory arrays. The inputs and outputs of MAGIC gates are stored within the memory array, in the same cells that are used for computation; that is, the entire computation is performed within the memory cells. MAGIC gates can form a complete logic structure using NOR/NOT operations, which means that any function can be broken down into a sequence of MAGIC operations.

MAGIC gates rely on initialization of their output cells to a specific logical state prior to the computation. Then, based on the input values, the output cell either switches or retains its value. In fact, MAGIC gates can skip the initialization step and use their output cells as an additional input variable. FELIX [9] presented a two-cycle MAGIC-based XOR gate. Both cycles use the same cell as an output, without a cell initialization between the cycles. By avoiding this initialization, the output value generated in the first cycle serves as an input for the second cycle, and the result of the second cycle overwrites the value stored in the output cell in the first cycle. By referring to the second cycle operation as a new MAGIC gate, *i.e.*, a MAGIC NOR gate without initialization, the idea of input overwriting with MAGIC can be leveraged.

Input overwriting, as referred in this paper, is the ability of a logic operation to replace one of its inputs with the operation result. We show that input overwriting provides additional functionality that improves the performance and extends the effective lifetime of the system.

The integration of input overwriting into the execution sequence of a full and complex combinational logic raises several challenges and difficulties. Existing PIM synthesis tools do not support the integration of input overwriting and therefore face these challenges. In this paper, we propose several principles and methods to address these challenges for different tools and algorithms. To demonstrate the impact of input overwriting in PIM, we introduce two eXtended MAGIC (X-MAGIC) gates with input overwriting capability, and modify an existing MAGIC synthesis tool by adding input overwriting support to it. We test two synthesis configurations, differing in their performance/area trade-off, using the modified synthesis tool and compare the results to a state-of-the-art synthesis process that does not support the integration of input overwriting.

This paper makes the following contributions:

- Identification of principles and development of methods to support input overwriting in PIM.

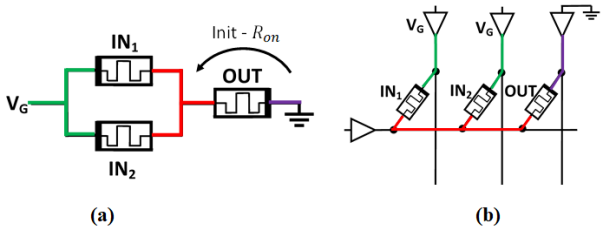

Fig. 1. (a) Schematic of a MAGIC NOR gate and (b) its mapping into a memory crossbar array. To perform a logical NOR operation using the gate, the output cell is initialized to R_{on}. Then, a voltage is applied at the negative terminals of input memristors and the negative terminal of the output memristor is grounded. The operation relies on a voltage divider between the input and output memristors.

- Proposing two new MAGIC gates that overwrite one of their inputs and store their outputs instead of it.
- Demonstration of the performance and effective lifetime benefits of the proposed gates.

The rest of the paper is organized as follows. Section II provides the background about MAGIC and the relevant synthesis tool. Section III introduces the concept of input overwriting, describes its general principles, and shows how to apply it to extend MAGIC. The methodology and evaluation are presented in Section IV, and Section V concludes the paper.

II. Background

The concept of input overwriting may be useful for a diversity of technologies, algorithms, and tools. In this work, we focus on the MAGIC technique and the synthesis tool called SIMPLER.

A. MAGIC

Memristor aided logic (MAGIC) [4] is a technique which uses memristors to perform logic operations, including logical NOR and NOT. Since NOR constitutes a functionally complete set, MAGIC gates can implement any combinational logic sequence. MAGIC gates work with the same data representation of the stored data in memristive memories, where logical '1' is represented as a low resistive state (R_{on}) and logical '0' is represented as a high resistive state (R_{off}). Fig. 1 shows the schematic of a single MAGIC NOR gate and its mapping into a memory crossbar. The operation of this gate (as well as the NOT gate) consists of two steps. At the first step, the output memristor is initialized to R_{on}. At the second step, a voltage is applied to the negative terminals of the input memristors, while the negative terminal of the output memristor is grounded. By applying the voltage, the resistance of the output memristor is changed depending on the ratio between the resistances of the input and the output memristors.

Operation of MAGIC gates is compatible with memristive memory crossbars, allowing for performance of single instruction multiple data (SIMD) operations. By selecting the same columns as inputs and output in multiple rows, all the selected rows behave as the same logic gate, but with different input values. Furthermore, the initialization stage can be performed simultaneously on multiple columns within all selected rows.

B. SIMPLER

To perform different logical functions with MAGIC, the MAGIC gates must be sequenced and then mapped into specific locations within the memory. SIMPLER [10] is a state-of-the-art synthesis and mapping tool for the conversion of combinational netlist files into PIM execution sequences, where each sequence can be executed within a single row in the crossbar array. The execution sequence created by SIMPLER is composed of MAGIC instructions, where each instruction includes the operation type (NOR or NOT), the location of its inputs, and the location where its result should be placed within the memory row. The tool also minimizes the number of memory cells participating in the computation.

III. Input Overwriting

As explained above, input overwriting is the ability of an operation to replace one of its inputs with its result, *i.e.*, the ability to overwrite an input with the operation output. Input overwriting introduces new functionality based on the same operation, which may improve the performance, area, and effective lifetime of the system. However, using input overwriting raises several limitations and implications. In this section, we discuss some of the challenges caused by input overwriting, and describe principles and methods to handle them. Then, we describe how to use input overwriting with MAGIC and how to integrate it in logic synthesis.

For our discussion, we describe combinational logic as a dependency Directed Acyclic Graph (DAG) [10]. The combinational logic is synthesized using an existing CMOS synthesis tool that generates a list of logic gates and their connections. For a given netlist, each logic gate is represented as a node (where the node is associated with the gate's logic operation), and a wire between logic gates is represented as an edge. The inputs of each gate are referred as its child nodes, while the gates that are connected to its output are referred to as its parent nodes. The number of parents a node has is termed as the *Fan Out* (FO) of the node. To perform the logic operation associated with a node, all the operations associated with its child nodes must be performed prior to the node operation, *i.e.*, the node depends on its children in term of execution order. Wires (*i.e.*, edges) describe true dependencies between the nodes they connect. The roots of each DAG are the outputs of the netlist, and the leafs of the DAG are the inputs of the netlist. Besides a logic operation, each node is associated with the operation value (its result). In this paper, an overwriting relation (dependency) between two nodes means that the result of the parent node operation **replaces** the original value that is associated with the child node.

To distinguish between overwriting dependencies and non-overwriting dependencies, we use two types of edges in the dependency DAG. The first edge type, named *regular edge*, represents a non-overwriting dependency. The second edge type, named *overwriting edge*, represents an overwriting relation between a child node and a parent node, namely, the value stored in the child node is overwritten by the parent operation. Fig. 2(a) shows the two different types of edges.

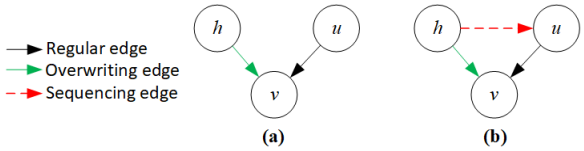

Fig. 2. DAG of three logic gates, demonstrating the different types of edges. Nodes u and h consume the output of node v. Node u is connected to v via a regular edge (black), and node h is connected to v via an overwriting edge (green). (a) Execution order impact. If h is executed before u, the original value of v is overwritten; hence it is invalid as an input of node u. (b) Use of a sequencing edge. To ensure u is executed before h, a sequencing edge (red) is added.

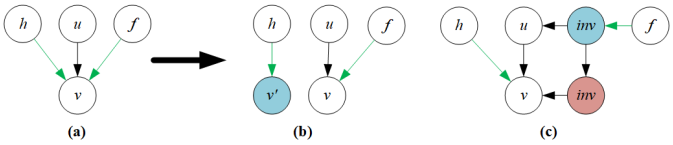

Fig. 3. Example of several overwriting parents. The nodes in the DAG are v, h, u and f. Node u is a parent of v connected via a regular edge (black). Nodes h and f are parents of v connected via overwriting edges (green). (a) Several overwriting parents. Without intervention, f and h will overwrite the value stored in v, where the first node that is executed will make the value invalid for the other. (b) The solution if v is not connected to any of its children via an overwriting edge. v is duplicated into v' (blue node), and the edge from h to v is replaced by an edge from h to v'. (c) The solution if v is connected to one of its own children via an overwriting edge. Two inverter nodes in a row (a buffer) are inserted between v and f ($f \rightarrow$ blue node $\rightarrow$ red node $\rightarrow v$). If u is associated with an inverter, u operates as the first buffer's inverter, and only the second inverter is inserted ($f \rightarrow$ blue node $\rightarrow u \rightarrow v$).

Note that in this work, a parent node can have, at most, one child connected to it via an overwriting edge.

A. Input Overwriting Principles

In this section, we describe several principles that an input overwriting DAG should follow, explain why these principles are needed, and describe methods to ensure that a given DAG is following these principles.

The first principle for working with input overwriting is to check that **each node has no more than one parent connected to it via an overwriting edge**. Otherwise, each one of its overwriting parents will try to overwrite the value stored in the node. Let h, u, f, and v be nodes in the DAG, as shown in Fig. 3(a). If h is executed before f, the value that is stored in v will be overwritten, hence one of inputs that belongs to f will contain an incorrect value. The same phenomenon will occur if the execution order of h and f is reversed. This case can be further generalized by having more than two parents connected to v via overwriting edges. Our method for overcoming this challenge is based on the relation between v and its children. If v is not connected to any of its own children via an overwriting edge, we duplicate v into a new node, v', where v and v' have the same children. Then, we choose one of v's overwriting parents and move the connecting overwriting edge to connect between v' and the parent instead, as shown in Fig 3(b). We repeat this process until v has only a single parent connected via an overwriting edge. If v is connected to one of its own children via an overwriting edge, the duplication of v will also require duplication of the overwritten child of v. That is, at least two nodes will be duplicated, and even more than two if v's child is also connected to one of v's grandchildren via an overwriting edge. Hence, instead of duplication, a buffer is inserted between v to all but one of its overwriting parents. Fig. 3(c) demonstrates the insertion of the buffer nodes.

Another principle to follow is that **the overwriting parent should be the last parent that is executed**. Assume there are two parent nodes with a shared child node, where the first parent node is connected to the child node via an overwriting edge, and the second node is connected via a regular edge, as shown in Fig. 2(a). If the overwriting parent node is executed before the other parent node, the value of the child will be changed, and hence, it will hold an incorrect value as an input for the second parent node. This ordering problem may become even worse if there are several parents that are connected to the same child via regular edges. The method to solve this issue consists of defining a third type of edge - a *sequencing edge*. The purpose of this edge is to ensure that the parent that is connected to the mutual child via an overwriting edge, is the last one to be executed. Sequencing edges create artificial dependencies among all the parents that are connected to the child via regular edges and the parent node that is connected to the child via an overwriting edge, as illustrated in Fig. 2(b). Sequencing edges are added only after it was ensured that each node has no more than one parent connected to it via an overwriting edge (as described in the first principle).

The third principle is that **sequencing edges must not create cycles in the DAG**. Let v, h, and u be nodes in the DAG, as shown in Fig. 4(a). The addition of the sequencing edge from h to u creates a cycle, since a path between u and h exists. The existence of this cycle contradicts the fact that the dependency graph is a DAG and creates a cyclic dependency. We call this problem the *cyclic dependency problem*. To avoid the *cyclic dependency problem*, we check each node for a connection to one of its parents via an overwriting edge. If an overwriting edge exists, we search for a path from any of the other parents (u) to the one which overwrites (h) the mutual child. If a path is found, we distinguish between the same cases as we did in the single overwriting parent check, as shown in Fig. 4(b) and Fig. 4(c).

The last principle is that **overwriting the inputs of the original netlist is prohibited**. If one of the input nodes of the netlist is connected to its parent via an overwriting edge, the execution of this parent will change the input value, and it will not be available to other netlists which use this input or as general data. Hence, it is necessary to protect the value of this input. To support this, we use an approach similar to those shown in Fig. 3 and Fig. 4.

We combine all the above methods into a single process, and apply it to the initial DAG. We name the entire process the *Pre-Processing Stage*. To analyze the time complexity of the *Pre-Processing Stage*, assume the number of nodes in the initial DAG is $|V|$, and the number of edges is $|E|$. The time complexity of assuring that each node has at most one parent connected to it via an overwriting edge, adding

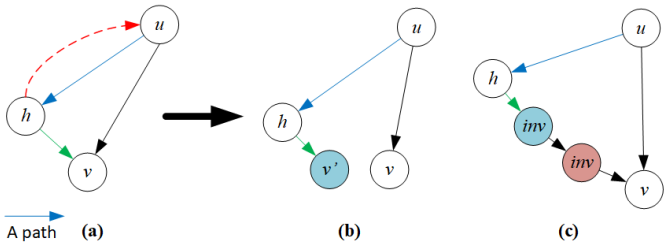

Fig. 4. The cyclic dependency problem and the proposed solution. The nodes in the DAG are v, h, and u. Node u is a parent of v and is connected to v via a regular edge (black), h is a parent of v and is connected to v via an overwriting edge (green). Let p be a path from u to h, represented by the blue edge. (a) A cyclic dependency. Adding a sequencing edge (red) from h to u creates a dependency cycle, composed of p and the edge. (b) The solution if v is not connected to any of its children via an overwriting edge. v is duplicated into v' (blue node), and the overwriting edge from h to v is replaced by an edge from h to v'. (c) The solution if v is connected to one of its own children via an overwriting edge. Two inverters (as a buffer) are inserted between v and h. If v has one inverter as a parent (red node), only one inverter is inserted (blue node).

the sequencing edges, and protecting the netlist's inputs, is $O(|V| + |E|)$. To solve the *cyclic dependency problem*, we use the *DFS* algorithm for checking the existence of paths to each overwriting node, which makes the time complexity of this method to be $O(|V| \cdot (|V| + |E|)) = O(|V|^2 + |V||E|)$. Hence, the time complexity of the entire *Pre-Processing Stage* is $O(|V|^2 + |V||E|)$.

B. Input Overwriting using MAGIC

As described in Section II-A, to perform a NOR/NOT operation using MAGIC, the output cell must be initialized to R_{on} prior to computation. FELIX [9] introduces a two-cycle MAGIC XOR gate, which uses the same output cell for both cycles without a re-initialization between the cycles. By avoiding this initialization, FELIX makes use of the input overwriting concept, but only as an intermediate computation step. FELIX does not show the potential of MAGIC-based input overwriting beyond small and handcrafted logic circuits, nor does it address the challenges introduced by input overwriting. FELIX does not discuss how to integrate input overwriting in a full synthesis flow. In FELIX, the intermediate value is unavailable as an input for other MAGIC gates, which may result in some re-computations. By defining new MAGIC-based gates which use the input overwriting capacity, applying the *Pre-Processing Stage* to any given combinational netlist that uses the new gates, and adding the relevant support to SIMPLER, we present a full synthesis flow for any function, that supports both the new and the old MAGIC gates.

Assume a MAGIC NOR operation is performed on inputs B and C, and the operation result is stored in an uninitialized cell, A. Table I lists the possible results for different input combinations. If the value of A before the computation is logical $'0'$ (R_{off}), the output value, *i.e.*, the value stored in A after the computation, marked as A^*, is logical $'0'$ (R_{off}) regardless of the values of B and C. The value that is stored in A does not change owing to the output memristor polarity and under the voltage constraints for proper operation [4]. Conversely, if the value of A prior the computation is logical $'1'$ (R_{on}), the gate operates as a standard MAGIC NOR gate. Overall, A can be viewed as an input of a gate that implements the $A \cdot (\overline{B + C})$ logic function, where the output value (A^*) and input A must share the same memory cell. In other words, the result value overwrites the value stored in A. Similarly, performing a MAGIC NOT operation with an uninitialized output cell implements the $A \cdot \overline{B}$ logic function. We named these overwriting MAGIC gates as X-MAGIC (eXtended MAGIC) gates.

TABLE I
X-MAGIC NOR TRUTH TABLE

A	B	C	Output (A^*)
0 (R_{off})	ϕ		0 (R_{off})
1 (R_{on})	0 (R_{off})	0 (R_{off})	1 (R_{on})
1 (R_{on})	0 (R_{off})	1 (R_{on})	0 (R_{off})
1 (R_{on})	1 (R_{on})	0 (R_{off})	0 (R_{off})
1 (R_{on})	1 (R_{on})	1 (R_{on})	0 (R_{off})

X-MAGIC gates provide additional capable logic on top of the standard MAGIC NOR/NOT gates; the enhanced functionality of the new gates may reduce the number of total gates needed to implement a given combinational function. To perform the same logic operation of X-MAGIC gates with MAGIC NOR/NOT, four clock cycles are required instead of the single clock cycle required in X-MAGIC. Hence, using X-MAGIC may lower the number of gates and reduce execution time. Furthermore, since X-MAGIC gates avoid the initialization of the output cell, and since fewer gates are used to execute a function, the number of write operations to the memory cells is also lowered, lengthening the effective lifetime of the system due to endurance limitations. The effective increase in lifetime is proportional to the relative reduction in the number of writes.

IV. Evaluation and Results

To evaluate the impact of X-MAGIC gates, we developed X-SIMPLER, a modified version of the Python-based implementation of SIMPLER to support the new gates. We also developed and integrated the *Pre-Processing Stage* as the first stage of X-SIMPLER flow, that is, before the synthesis and mapping stages. We examined the outputs produced by X-SIMPLER by comparing the performance (execution latency), area (required memory row size), and effective lifetime (correlates with the number of write operations per cell) of each tested benchmark to those of the of SIMPLER (using only NOR/NOT gates). In this paper, we evaluated the impact of the $A \cdot (\overline{B + C})$ gate only, since its functionality includes the functionality of the second X-MAGIC gate. This section describes the evaluation methodology and results.

A. Environment and Methodology

Synthesis constraints and optimization. The first stage of the X-SIMPLER flow is synthesis based on a standard synthesis flow (typically used for CMOS logic). To generate a netlist that contains X-MAGIC gates, the X-MAGIC gate was modeled as a three-input gate. A specific input port was

defined as the overwriting port (corresponds to A). Since the number of gates directly determines the latency, we attempted to reduce the number of nodes added during the *Pre-Processing Stage* by using the optimization capabilities of the synthesis tool. The synthesis was optimized for area, where all gates were assigned equal area. To ensure that each gate drives, at most, one overwriting input port, we use Fan-Out (FO) constraints during the synthesis process through the gates definition file. The X-MAGIC overwriting input port was assigned a high FO load value, while all the other input ports (whether they belong to an X-MAGIC or NOR/NOT), were assigned a low FO load value. The output ports of all gates were assigned a FO value that ensured that no more than a single overwriting input port will be connected to each gate output.

CMOS synthesis tool. SIMPLER used the ABC synthesis tool (ABC) [11]. Since *ABC* cannot apply any FO constraints, this work uses the *Synopsis Design Compiler* (DC) [12] as the synthesis tool. The gates definition file used by DC is written in *Liberty* format. To make a fair comparison, all workloads [13] were re-synthesized using DC. For a given set of design constraints, DC makes the "best-effort" for applying them. The constraints defined in the *Liberty* file may therefore be violated in the netlist created by the tool. The *Pre-Processing stage* was built in a way that checks if violations exist, and solves these violations during its different sub-stages.

Synthesis configurations. We examined two synthesis configurations for the integration of the X-MAGIC gates (each configuration is defined by a different *Liberty* file). The first configuration allows each gate to drive, at most, a single X-MAGIC node via an overwriting port (edge) and to drive several additional gates via regular ports (including X-MAGIC gates, to their regular ports), as shown in Fig. 5(a). In the second configuration, each node can only drive either a single overwriting port or several regular ports, as shown in Fig. 5(b). The configurations are named *Different-Edge Types* (DET) and *Same-Edge Types* (SET), respectively.

Benchmarks, baseline, and comparison. To evaluate the impact of the X-MAGIC gates in both configurations, we used the EPFL benchmark suit [13]. We applied SIMPLER on the re-synthesized files (without X-MAGIC), where the results serve as the baseline for the comparison between the two X-MAGIC synthesis configurations. To evaluate the potential effective lifetime improvement, we assumed the operations are uniformly distributed among all cells within each memory row (as proposed in, *e.g.*, *RRAM Endurance Resiliency* [14]). Under this assumption, we measured only the reduction in the number of write operations for each tested benchmark. To assess the impact of X-MAGIC on area, we measured, for each configuration, the minimum row size needed for the computation of the different benchmarks. To make a fair latency comparison between SIMPLER and X-SIMPLER (for both SET and DET), the memory row size each tool uses should be the same. For the comparison between each X-MAGIC configuration (SET **or** DET) and SIMPLER, we

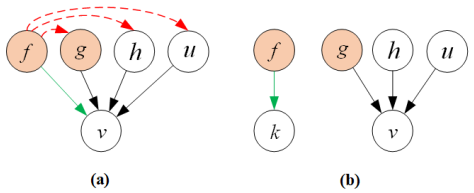

Fig. 5. The two tested synthesis configurations. f, g, h, u, k, and v are nodes in the DAGs. Nodes h, u, k, and v are regular MAGIC nodes (*i.e.*, NOT or NOR), and f and g are X-MAGIC nodes (marked in orange). (a) Different-Edge Types (DET). Each child can be connected to both regular and overwriting edges. (b) Same-Edge Types (SET). Each child can be connected to only a single edge type. After the *Pre-Processing Stage*, the child node can be connected to several regular edges, or to a single overwriting edge.

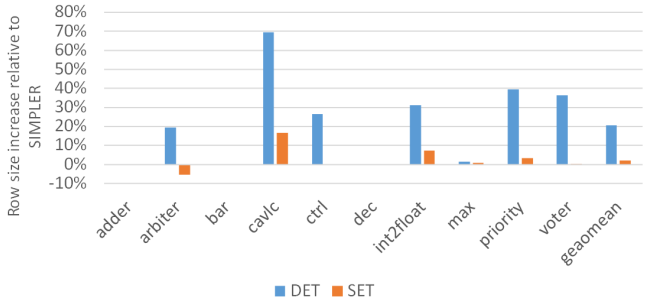

Fig. 6. Row size increase relative to SIMPLER (lower is better).

set the size to the highest of the two minimum row sizes SIMPLER and X-SIMPLER require, where SIMPLER serves as the baseline. To make a fair latency comparison between SET and DET, we set the row size to the highest of the three minimum sizes SIMPLER and X-SIMPLER (SET and DET) require, where SIMPLER serves as the baseline.

B. Results

Area. SIMPLER uses a heuristic technique to reuse cells and reduce the row size. This heuristic is not fully compatible with the case where the execution order influences the correctness of the logic. Hence, X-MAGIC increases the row size. Fig. 6 shows the increase in the minimum required row size for each configuration compared to SIMPLER minimum row size. The geomeans of DET and SET configurations, relative to SIMPLER, are 20.5% and 2.1%, respectively. The increase in the row size for SET is relatively low, and is even lower than the baseline in one case (arbiter). Conversely, the DET configuration incurs a non-negligible area increase. The zero row-size increase in adder and dec stems from the fact that these two benchmarks do not use X-MAGIC gates at all.

Latency. A netlist consisting of both X-MAGIC gates and standard MAGIC gates has fewer gates than a netlist consisting of MAGIC gates only. Since the latency is proportional to the number of gates, the latency is lower when X-MAGIC gates are used. Fig. 7 shows the latency relative to the baseline. Note that for benchmarks which do not use X-MAGIC gates (adder and dec), there is no improvement. The comparisons between each configuration and its SIMPLER baseline are shown in blue for DET and in orange for SET. The geomean latency improvements are 18% for DET and 16.2% for SET. We select the row size to be the maximum between the sizes

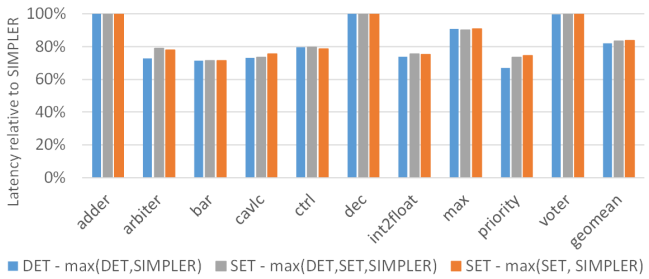

Fig. 7. Latency relative to SIMPLER (lower is better).

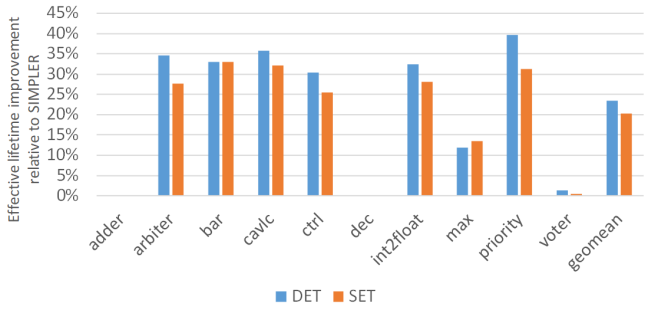

Fig. 8. Effective lifetime improvement relative to SIMPLER, measured as reduction in the number of write operations (higher is better).

of SIMPLER and the compared SET or DET configuration. To compare between DET and SET, we set the same row size for both configurations. Since the highest size is always the size in DET, only SET is re-executed using the row size of DET. The results for SET with this row size are shown in gray, where the geomean latency improvement is 16.5%.

Lifetime. We approximate the lifetime improvement by counting the number of write operations associated with each benchmark. We count each occurrence of a standard MAGIC gate (NOR/NOT) twice, once for the output initialization and once for the gate operation. Each X-MAGIC gate is counted once, as they do not require initialization. Fig. 8 shows the effective lifetime improvement (measured as reduction in the number of write operations) relative to SIMPLER. The geomean improvement for DET is 23.4% and for SET is 20.2%. This shows that both configurations fulfill our expectations for a nice improvement in the effective lifetime of the system.

The differences between the DET and SET configurations relative to SIMPLER are clear. Both configurations significantly improve latency and reduce number of write operations, but need a larger row. Among the DET and SET configurations, when both are allocated the same row size, DET is slightly better than SET in both latency and number of writes, but DET needs a larger minimal row size to operate. Based on these findings, it is reasonable to conclude that DET is likely a better choice in general, unless area is so scarce that saving several cells makes a difference.

Finally, the computational complexity of the *Pre-Processing stage* is $O(|V|^2 + |V||E|)$, higher than SIMPLER computational complexity of $O(|V| + |E|)$. This increase is less significant in practice since (1) the modified tool is still quite fast, taking less than one minute to process the largest benchmark (which contains over $14K$ gates), (2) the *Pre-Processing stage* is only a small part of the entire synthesis flow, hence, its longer runtime does not significantly affect the entire synthesis runtime.

V. Conclusion

This paper presents several principles and methods for working with input overwriting in PIM. It defines X-MAGIC, two MAGIC-based logic gates which overwrite one of their inputs. X-MAGIC gates implement the $A \cdot (\overline{B + C})$ and $A \cdot \overline{B}$ logic functions. In both functions, the operation result overwrites the same memory cell used to store the input value A. To demonstrate the impact of input overwriting on PIM, we added support for X-MAGIC gates in the synthesis of combinational logic for PIM. The results show a decent improvement in the latency and the effective lifetime, while the minimal area is slightly higher in one of the configurations. We examined two X-MAGIC synthesis configurations, and our experimental results show a geomean latency improvement of over 16.5% and a geomean improvement of over 20% in the effective lifetime for the same row size as the baseline.

Acknowledgment

This research is supported by the ERC under the European Unions Horizon 2020 Research and Innovation Programme (grant agreement no. 757259).

References

[1] Pedram *et al.*, "Dark Memory and Accelerator-Rich System Optimization in the Dark Silicon Era," *IEEE Design Test*, vol. 34, pp. 39–50, April 2017.

[2] L. Chua, "Memristor-The Missing Circuit Element," *IEEE Transactions on Circuit Theory*, vol. 18, pp. 507–519, September 1971.

[3] C. Xu *et al.*, "Overcoming the Challenges of Crossbar Resistive Memory Architectures," *2015 IEEE 21st International Symposium on High Performance Computer Architecture (HPCA)*, pp. 476–488, February 2015.

[4] S. Kvatinsky *et al.*, "MAGIC-Memristor-Aided Logic," *IEEE Transactions on Circuits and Systems II: Express Briefs*, vol. 61, pp. 895–899, November 2014.

[5] Borghetti *et al.*, "'Memristive' Switches Enable 'Stateful' Logic Operations via Material Implication," *Nature*, vol. 464, pp. 873–876, April 2010.

[6] S. Kvatinsky *et al.*, "Memristor-Based Material Implication (IMPLY) Logic: Design Principles and Methodologies," *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, vol. 22, pp. 2054–2066, October 2014.

[7] Linn *et al.*, "Beyond von Neumann—Logic Operations in Passive Crossbar Arrays Alongside Memory Operations," *Nanotechnology*, vol. 23, July 2012.

[8] Y. Levy *et al.*, "Logic operations in memory using a memristive Akers array," *Microelectronics Journal*, vol. 45, no. 11, pp. 1429 – 1437, 2014.

[9] S. Gupta *et al.*, "FELIX: Fast and Energy-Efficient Logic in Memory," in *2018 IEEE/ACM International Conference on Computer-Aided Design (ICCAD)*, pp. 1–7, 2018.

[10] R. Ben-Hur *et al.*, "SIMPLER MAGIC: Synthesis and Mapping of In-Memory Logic Executed in a Single Row to Improve Throughput," *IEEE TCAD*, July 2019.

[11] A. Mishchenko, "ABC: A System for Sequential Synthesis and Verification," *Berkeley Logic Synthesis and Verification Group*, 2012.

[12] H. Bhatnagar, *Advanced ASIC Chip Synthesis: Using Synopsys' Design Compiler and PrimeTime*. USA: Kluwer Academic Publishers, 1999.

[13] L. Amarù *et al.*, "The EPFL Combinational Benchmark Suite," in *Proceedings of the 24th International Workshop on Logic Synthesis (IWLS)*, 2015.

[14] M. M. Sabry Aly *et al.*, "The N3XT Approach to Energy-Efficient Abundant-Data Computing," *Proceedings of the IEEE*, vol. 107, no. 1, pp. 19–48, 2019.