SIMPLE MAGIC: Synthesis and In-memory MaPping of Logic Execution for Memristor-Aided loGIC

Rotem Ben Hur*, Nimrod Wald, Nishil Talati, and Shahar Kvatinsky
Andrew and Erna Viterbi Faculty of Electrical Engineering
Technion – Israel Institute of Technology
Haifa 3200003, ISRAEL
*rotembenhur@campus.technion.ac.il

Abstract—**This paper presents a novel approach for designing and implementing in-memory logic operations. The uniqueness of this work is the development of SIMPLE, a framework that optimizes the execution of an arbitrary logic function, while considering all the constraints involved in performing it within a memristive memory. SIMPLE automatically generates a defined sequence of atomic memristor-aided logic NOR operations, whose implementation can be facilitated efficiently within the memory. Motivated to overcome the memory-CPU bottleneck, this approach designs an optimal solution in terms of performance by exploiting the parallelism of the memristor-aided logic NOR gates. SIMPLE achieves performance speedups of 1.94x compared to a previous work and 1.48x compared to a naïve optimization based on standard synthesis tools.**

*Keywords—**Memristor, memristive systems, logic design, MAGIC, mMPU, von Neumann architecture, logic synthesis.***

I. Introduction

Over the last several decades, the rate of improvement in processors has exceeded that of memory by several orders of magnitude. The separation between the memory and CPU in von Neumann architecture and the need to transfer data between them have created the primary performance and energy bottleneck in modern computing systems. This bottleneck is known as the *memory wall*. Methods for alleviating the memory wall have been widely explored by researchers, using different techniques to reduce data transfer between the CPU and memory, usually by exploiting data locality in the memory system. Processing data within the memory itself seems like the ultimate way to break the von Neumann separation.

Performing pure in-memory computing is only possible when the same physical entities are used for both memory (*i.e.*, data storage) and logic (*i.e.*, data processing). Conventional memory architectures, such as SRAM, DRAM and Flash, do not offer this capability. Emerging nonvolatile memory technologies, on the other hand, have the capability to perform logic operations in certain conditions. These technologies include RRAM, PCM, STT MRAM and others. For simplicity, we refer to all of them as *memristors*. The memristor is a passive element with numerous promising features, such as low power consumption, CMOS fabrication compatibility, high density, and good scalability [1], [2]. An attractive approach for performing logic within a conventional memristive memory array is *stateful logic*, where the logical states are represented by resistance. The memory cells are used to construct logic gates, where the inputs and outputs are, respectively, the states of specific memristors before and at the end of the computation. This paper considers an improved stateful logic family called *Memristor-Aided loGIC* (MAGIC) [3] that outperforms previously proposed stateful logic families [4]. The key idea behind MAGIC in-memory is to use it to execute NOR operations, which can be used as the basis for performing any desired computation.

A recently proposed architecture, where the conventional DRAM memory is replaced with a *memristive Memory Processing Unit* (mMPU), is described in [5]. mMPU is a memory architecture with processing capabilities, based on MAGIC NOR and NOT operations. Although it has the potential to perform general purpose computations, current methods require that the sequence of MAGIC operations required to implement the desired logical function in-memory be manually designed [6]. Obviously, this approach is neither general nor optimal, since it does not guarantee a full utilization of in-memory computation advantages, especially the parallelism offered by MAGIC-based operations [4].

This paper presents SIMPLE, a general framework that allows the implementation of arbitrary logical functions within the memory in an optimal manner. To the best of our knowledge, this is the first synthesis tool that makes it possible to map logical gates to specific memory cells of a conventional memristive memory. Hence, this work creates, for the first time, an end-to-end in-memory implementation method for any logical function. For that purpose, we develop an automatic tool that combines the synthesis of MAGIC-based functions with an optimal mapping into the memristive memory. This is done while minimizing the latency of the entire computation by finding an optimal division of the logical function into small execution steps. We define and solve an optimization problem to get the best possible sequence of NOR operations that maximally exploits the mMPU for a certain computing task. This optimization offers 89%, 48% and 94% latency improvements on average as compared to, respectively, the original circuit implemented with NOR and NOT gates, a naïve approach synthesized with standard CMOS synthesis tools with NOR and NOT gates, and previous work which synthesizes by

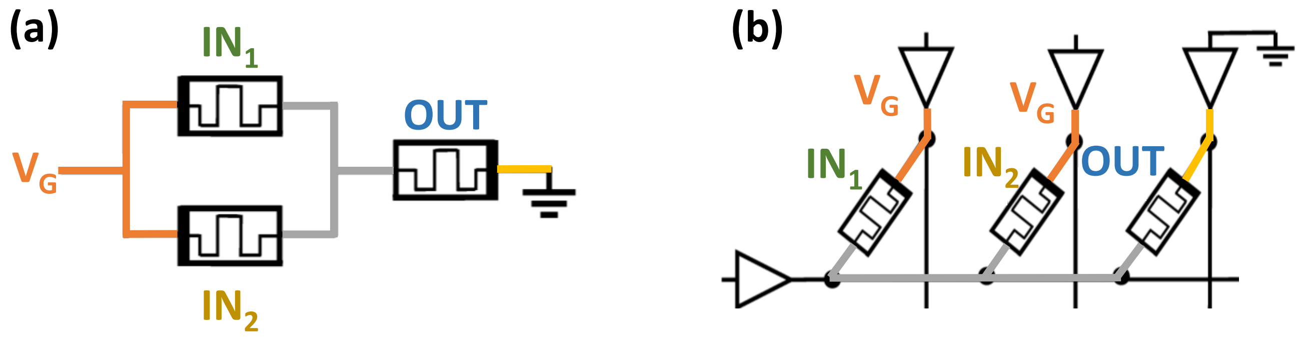


Figure 1. Schematic of (a) MAGIC NOR gate and (b) MAGIC NOR gate within a memristive memory array. *IN_1* and *IN_2* are the inp memristors and *OUT* is the output memristor. A single voltage *V_G* is applied to perform the NOR operation [3].

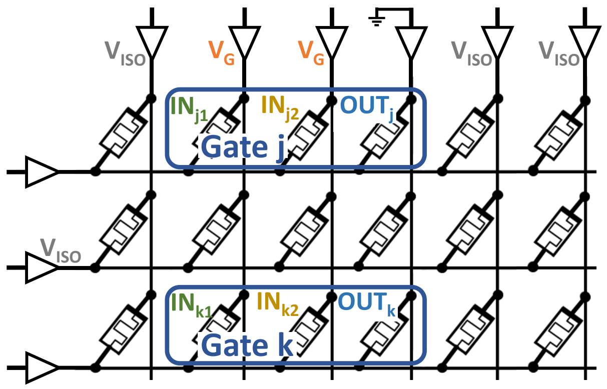


Figure 2. Parallel MAGIC NOR execution of gates *j, k*. Parallel execution of gates requires alignment of their inputs and output. Gates *k* and *j* are aligned by their columns. Since the memory is symmetric, the gates can also be aligned by rows. The operation is performed within the memristive memory array by applying *V_G* to the columns of the input memristors, ground to the column of the output memristor, and *V_{ISO}* to isolate unselected columns and rows. The operation takes a single clock cycle, regardless of the number of gates executed in parallel.

merely dividing into execution steps, without considering the mapping into the memory cells [7]. Therefore the authors evaluations in [7] are optimistic (*i.e.*, without considering the exact data location and the overhead of moving data to the appropriate location in order to process it) and our improvement over their results is probably much more significant than 112%.

II. Background and Related Work

Over the last years, several methods for synthesizing specific logical operations for memristor-based implementation were developed, only a few of which are suitable for in-memory computations. However, almost all work in this field focuses merely on the synthesis of the logical function, and not on its mapping into the actual memory cells. Furthermore, most synthesized functions cannot be executed within the memory as is, but require additional cycles (*e.g.*, *copy* and *read* operations) to achieve the desired functionality. Since other works do not offer a full implementable framework, their computation latency cannot be correctly evaluated, and no relevant comparison can be performed. Nevertheless, for completeness we describe the previous achievements in this field, alongside their advantages and disadvantages. In this section, we describe previous work on synthesis with memristors and discuss approaches for executing true in-memory processing using MAGIC operations.

A. Previous Work

Previous attempts to synthesize logic functions using memristors have been focused mostly on non-stateful logic techniques. Therefore, in-memory execution of these methods is restricted. Logic techniques that combine CMOS and memristors [8], [9] are not suitable for in-memory computing, since CMOS logic has to be added within the memory array. Another synthesis technique is based on the FBLC method [10], which requires disabled memristors (permanently in the high resistance state) [11]. This technique is thus also unsuitable for execution within fully operational memristive memories. Even though using a dedicated memory allows computations to be performed with the FBLC method, the processing tasks must be determined before fabrication, and cannot be dynamically chosen as in stateful logic. In addition, the area utilization of this type of processing is low, since the percentage of disabled memristors in the computation area is high.

Synthesis of in-memory computing includes both optimizing the performance and mapping the operations into specific memory cells within the memristive memory array. One synthesis approach is based on the majority function [12]–[14] using *Majority-Inverter Graphs* (MIGs), where memristors execute majority and negation functions. MIGs show good results in logic optimization in terms of the number of levels, as compared to data structures such as *Binary Decision Diagrams* (BDDs) and *And-Inverter Graphs* (AIGs) [15]. However, the number of computational steps presented in [12]–[14] is estimated from the number of MIG levels. This estimation is based on the assumption that all Majority gates from the same level are executed in parallel, which is usually not feasible when physical mapping into a crossbar is taken into account. Additionally, the output of the majority gate is the logical state of the memristor (represented by its resistance) at the end of the computation, whereas the inputs of the majority gate are the voltages applied to the rows and columns. Hence a read operation is required between every two chained majority

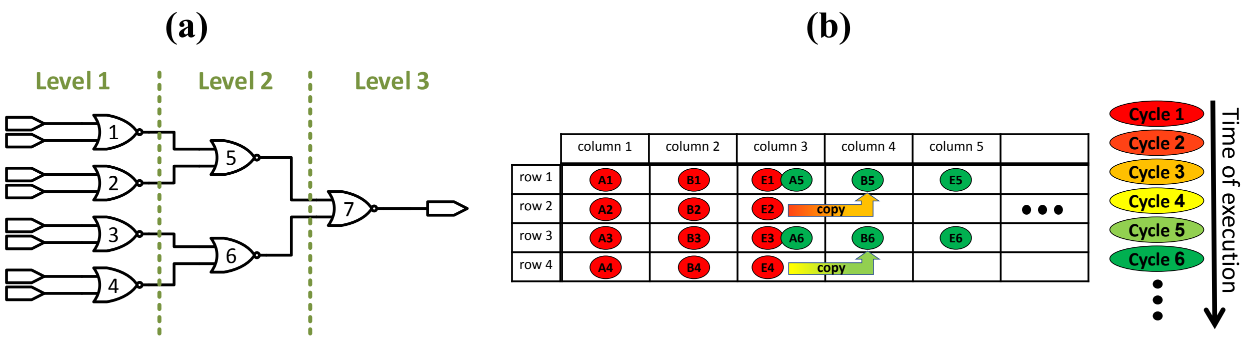


Figure 3. (a) CMOS NOR netlist divided into levels, and (b) conventionally synthesized netlist mapped to a memristive memory array. A_i, B_i are inputs and E_i is the output of gate i. The table represents locations in the memory array and the colors represent the corresponding cycle number. Each copy operation takes two clock cycles.

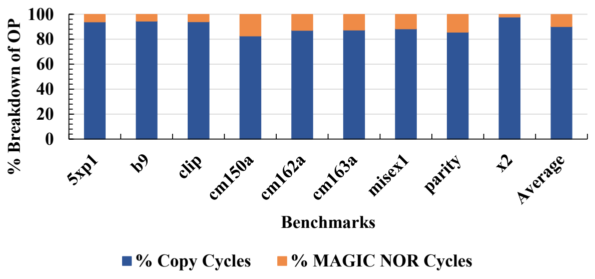


Figure 4. Breakdown of execution steps (OP) to number of MAGIC NOR and copy cycles, when using conventional synthesis flow.

gates, and a complex controller for executing in-memory logic has to be developed.

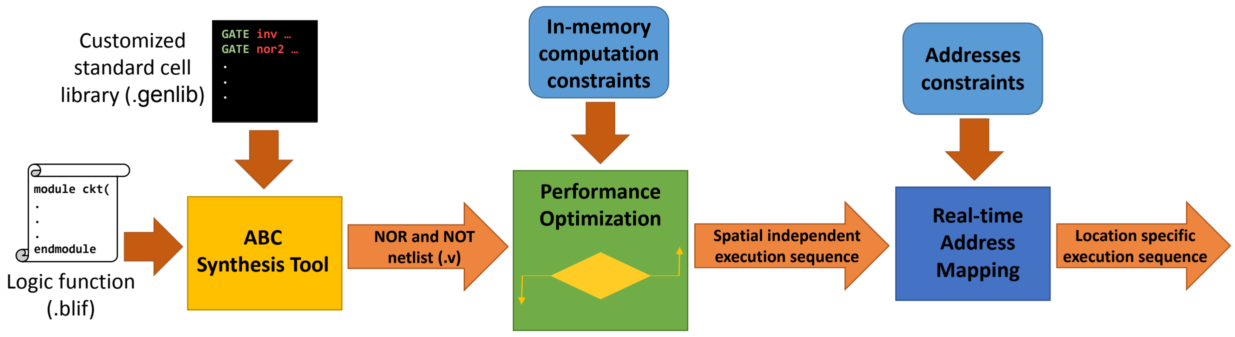
Another approach is using stateful logic families, such as *material implication* (IMPLY) together with *false* operations [16], [17]. Since IMPLY is an unconventional logic gate, new op timization methods for lowering the latency of computation must be developed. Previous attempts have been mostly manual and without considering the exact mapping within memory [18]–[22]. Ignoring the mapping prevents full utilization of stateful logic. Efforts towards full synthesis of logic functions using IMPLY have been made in [23], where similar to the MIG case mentioned above, the number of steps is erroneously estimated by the number of *Or-Inverter Graph* (OIG) levels. The sole work which comes close to estimating the actual number of in-memory execution steps is [7], where the IMPLY-based logic synthesis is done using a BDD. Each node of the BDD is mapped to a 2-to-1 multiplexer (MUX), implemented with an IMPLY gate. In their estimation of the number of execution steps, the authors exclude the numerous copy cycles required for in-memory parallel execution with IMPLY. Therefore, the improvement of SIMPLE as compared to [7] is more significant than is evident from a plain comparison with their results. Furthermore, although their method is based on parallel execution, the size of a BDD depends exponentially on the number of inputs, thus limiting the performance of this approach for large benchmarks. However, since they actually produce the sequence of operations executed within the memory, while determining which gates are executed in parallel, then this is the only fair comparison to our work.

B. *Memristor-Aided LoGIC (MAGIC)*

Memristor-Aided loGIC (MAGIC) [3], a stateful logic family that outperforms the previously proposed stateful logic families, has been recently proposed. In MAGIC, only a single voltage V_G is used to perform a NOR logic operation, and the inputs and output memristors are separated, as shown in Figure 1 (as opposed to IMPLY, where two different voltages are required, and one of the input memristors is also the output memristor, causing that input to be overwritten by the execution). Additionally, MAGIC gates do not require additional devices to perform the operation (unlike IMPLY, which requires an additional resistor for each row). Furthermore, because NOR is a standard logical function, executing functions based on NOR operations is simple and straightforward. Since NOR is a complete logic function, a MAGIC NOR operation is sufficient to execute any Boolean operation. Hence, MAGIC NOR may be used as the basis for performing all desired processing within memory by dividing the desired function into a sequence of MAGIC NOR operations. These basic NOR operations are executed one after the other using the memory cells as computation elements. Another advantage of MAGIC is its ability to perform logic operations in parallel on sets of data. The crossbar array is structured such that applying the operating voltage V_G on any two selected columns and grounding a third column will result in NOR operations being performed on all rows on which V_{ISO} is not applied (i.e., on rows that are not isolated). The schematic of a MAGIC gate operation, performed over gates arranged in different rows and aligned in columns within a memristive memory, is shown in Figure 2. Note that due to the symmetry of memristive crossbar arrays (*i.e.*, transpose memory), performing NOR operations on column vectors is feasible in a similar manner.

III. SYNTHESIS FOR MAGIC WITHIN MEMORY

In conventional CMOS logic synthesis flow, an input netlist representing a Boolean function is first divided into multiple levels, based on the data dependencies, as shown in Figure 3a. The input values of each gate in any level must be produced in one of the previous levels. Therefore, there are no input dependencies among gates from the same level. For example,



Customized standard cell library (.genlib)
GATE inv …
GATE nor2 …
module ckt(
endmodule
Logic function (.blif)
ABC Synthesis Tool
NOR and NOT netlist (.v)
In-memory computation constraints
Performance Optimization
Spatial independent execution sequence
Addresses constraints
Real-time Address Mapping
Location specific execution sequence

Figure 5. Proposed logic synthesis flow. The desired logic function is synthesized using ABC for NOR and NOT gates and then optimize specifically for MAGIC within memory, generating a general mapping and a sequence of operations. The general execution is mapped t specific cells in real-time, based on the temporary state of the mMPU.

outputs of gates in Level 1 (*Ei* where *i*=1,2,3,4) are the inputs of gates in Level 2 (*Aj*, *Bj*, where *j*=5,6). Hence, the execution of gates 5,6 can only begin after gates 1,2,3,4 finish their operation. Furthermore, in CMOS implementation, each level can be executed in parallel.

In the case of in-memory computing, the location of data is an added constraint. As shown in Figure 3b, for parallel MAGIC NOR execution the inputs must be aligned in the same columns. Under the assumption that the inputs of Level 1 are already aligned, they produce outputs *Ei* (where *i*=1,2,3,4) in column 3 in a single clock cycle, as shown. For parallel execution in Level 2, the inputs must be realigned. This is done by copying the desired operands to the same columns in two cycles using two MAGIC NOT operations.

In the case of larger functions, these copy operations can significantly degrade the performance of the system. To find the cost of copy operations, we conducted experiments for an in-memory computing approach on various benchmarks [24] using conventional synthesis flow. Figure 4 shows the breakdown of the number of execution steps (OP) as compared to the number of actual MAGIC NOR operations and the number of copy cycles for aligning the data. We can conclude from the figure that, on average, 90.2% of the execution time is spent on data arrangement. Hence, the conventional logic synthesis flow cannot be directly adopted for memristive in-memory computing.

To overcome this limitation, we require a novel method for synthesis and mapping of a given logic function that does not waste energy and time on unnecessary copying operations. In this paper, we propose a flow for synthesizing logical functions called SIMPLE. SIMPLE takes a desired function and automatically constructs and optimizes the required sequence of MAGIC NOR operations. This sequence is optimized specifically for in-memory MAGIC execution, while exploiting the parallel processing capability offered by memristive memories.

A. Overall Logic Synthesis Flow

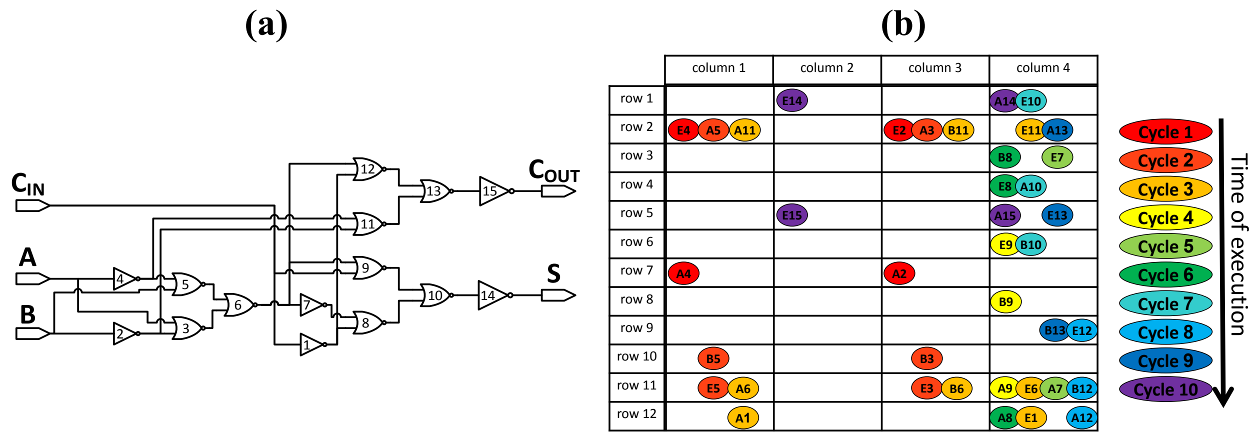
In the proposed synthesis approach, a desired logic function represented in Berkeley Logic Interchange Format (BLIF) is first converted into a NOT and NOR netlist representation, which is then area optimized using a modified ABC tool [25]. The produced netlist is, however, not optimal for in-memory logic since standard synthesis tools, such as ABC, are designed to minimize the area (number of gates) without considering the location and timing constraints of the computation steps, which are critical for efficient logic within memory.

The NOR and NOT netlist is mapped to a memristor-based memory, and the received mapping provides the optimal (lowest) latency, as detailed in Section IV. For a specific execution, the mapping is reshuffled in real time according to the addresses of the inputs and outputs of the desired function, in accordance with the specific occupancy of the memory at the time of execution. The synthesis flow is shown in Figure 5.

B. Real-time Operation and Control Signals

The mMPU architecture relies on having a set of predetermined functions that have been synthesized and stored in the mMPU controller [26]. The stored function set includes a general mapping of the inputs, outputs, and intermediate values to locations within a memory array, along with the relative number of steps in which each NOR operation is performed. When one of these functions is invoked, an exact real-time mapping of rows and columns is carried out to adjust the general locations to the exact locations of inputs and desired output, while considering the exact state of the memory (*e.g.*, stored data that cannot be overwritten, addresses of the input data and result).

After the real-time mapping, the operations in the set are translated by the mMPU controller into a set of control signals applied to the rows and column decoders of the memory. In other words, the inputs of the function are moved to the desired locations by the mMPU controller, if needed. Then the control signals compute the intermediate results step by step (where aligned gates are executed in parallel during the same clock cycle), and the last step yields the results in the desired address. The real-time mapper was not implemented here but will be developed in future work.



	column 1	column 2	column 3	column 4
row 1		E14		A14 E10
row 2	E4 A5 A11		E2 A3 B11	E11 A13
row 3				B8 E7
row 4				E8 A10
row 5		E15		A15 E13
row 6				E9 B10
row 7	A4		A2	
row 8				B9
row 9				B13 E12
row 10	B5		B3	
row 11	E5 A6		E3 B6	A9 E6 A7 B12
row 12	A1			A8 E1 A12

Figure 6. Single-bit full adder. (a) The netlist created by ABC, and (b) the optimized mapping and timing from the synthesis tool, achieving latency of 10 clock cycles. *Ai*, *Bi* are inputs and *Ei* is the output of gate *i*. The table represents locations in the memory array (using a memory array with 12 rows and 4 columns) and the colors represent the corresponding clock cycle.

IV. Optimization for Efficient Mapping Into Memristive Memories

To minimize the latency using SIMPLE, we define and solve an optimization problem for in-memory MAGIC execution. The optimization problem is described in this section, and h as been implemented using the AMPL modeling language [27] and using the IBM ILOG CPLEX Optimization Studio solver [28].

Since each input and output (I/O) of the logic gates is stored in a memory cell, each I/O is assigned to a coordinate (*i.e.*, location in the array). The optimization tool checks all possible execution patterns with different mappings (locations) of the I/Os of all gates, while considering the specific constraints of executing MAGIC within memory. The output of the synthesis tool is an optimal mapping with minimal latency. In this section, we describe the constraints for parallel processing within a memristive memory and the other major assumptions and constraints for the optimization problem.

A. Parallelism

The primary advantage of executing MAGIC within memory is its ability to execute numerous NOR operations simultaneously on different rows or columns, as described in Section B. Wisely exploiting the parallelism capability may significantly improve the computation performance.

In this section, a logic gate j is defined by the following variables:

- $\left(\left\{R_{A_j}, C_{A_j}\right\}, \left\{R_{B_j}, C_{B_j}\right\}, \left\{R_{E_j}, C_{E_j}\right\}\right)$ - Location (coordinates of memory cells) of the inputs A_j, B_j and the output E_j of the gate.
- T_j - Clock cycle (step) in which the gate is executed.

To perform parallel processing, the corresponding inputs and output of different gates should be aligned within the memory. Formally, gates $g_1, g_2, \ldots, g_k$ can be executed in parallel on multiple rows (meaning $T_1 = T_2 = \cdots = T_k$) iff $A_{i1}, B_{j2}, E_l, \forall i, j, l = 1, \ldots, k$ are located, respectively, in columns $C_n, C_m, C_r\ \forall n \neq m \neq r$ and $A_{i1}, B_{i2}, E_i,\ \forall i = 1, \ldots, k$ are in the same row. When using a transpose memory [4], parallel execution on multiple columns is also possible when switching between the rows and columns. As illustrated in Figure 2, gates k, j are aligned, since IN_{j1}, IN_{j2}, OUT_j respectively share columns with IN_{k1}, IN_{k2}, OUT_k, and the output and inputs of each gate share rows. Therefore, the execution of gates k, j can be performed at the same clock cycle.

Applying a voltage to a specific column affects all executed gates simultaneously. Hence, two NOR gates with a different number of inputs (*e.g.*, two-input NOR and NOT) cannot be executed in the same cycle. For simplicity, we define the problem merely for two-input NORs, and execute each NOT gate by forcing one input to '0', as $NOT(X) = NOR(X, 0)$.

B. Latency Optimization Problem

The problem of optimizing an in-memory Boolean function for minimal latency has two degrees of freedom: the locations of data and the execution cycle of different logic gates. The execution of a given Boolean function is finished when the operations of all gates are completed. Thus $max_j(T_j)$ is the latency of a specific mapping, where $0 < j \leq \#gates$. Therefore, the latency (in clock cycles) of the best mapping is the minimum latency out of all different mappings:

$$Latency_{best\ mapping} = min\left\{\max_j T_j\right\}, \quad 0 < j \leq \#gates. \tag{1}$$

The legal mappings are limited by location, connectivity and timing, and are restricted by the following constraints:

1) Location constraints:

- **Every I/O has to be mapped to a memory cell**. Thus, the coordinates of each I/O are limited by the physical size of the memory. Formally, for a memory array of the size $\boldsymbol{Row_{num}} \times \boldsymbol{Col_{num}}$, this constraint is

$$\forall \mathbf{x_j} \in \{\mathbf{A_j}, \mathbf{B_j}, \mathbf{E_j}\}: \left(\mathbf{0} < \boldsymbol{C_{x_j}} \leq \boldsymbol{Col_{num}}\right) \cap \left(\mathbf{0} < \boldsymbol{R_{x_j}} \leq \boldsymbol{Row_{num}}\right). \tag{2}$$

TABLE I. EXPERIMENTAL SYNTHESIS RESULTS.

Benchmarks	Original Netlist			ABC	SIMPLE MAGIC			Chakraborti *et al.* [7]	
	In	***Out***	***Gates***	***Gates***	***OP***	***Mem***	***Area***	***OP***	***Mem***
5xp1	7	10	171	112	97	142	315	73	84
clip	9	5	271	152	136	184	444	89	120
cm150a	21	1	88	62	51	87	189	127	56
cm162a	14	5	64	60	46	92	186	102	46
cm163a	16	5	63	61	45	95	183	116	42
misex1	8	7	93	78	45	112	294	69	83
parity	16	1	76	76	37	107	240	113	23
x2	10	7	98	68	36	86	168	80	60

- **Two different outputs cannot be placed in the same memory cell** (whereas the same input may be used for two different gates, and therefore their inputs share coordinates). Formally,

$$\forall \boldsymbol{E_k}, \boldsymbol{E_j}\colon \left(\boldsymbol{C_{E_j}} \neq \boldsymbol{C_{E_k}}\right) \cup \left(\boldsymbol{R_{E_j}} \neq \boldsymbol{R_{E_k}}\right). \tag{3}$$

  In such a configuration, reuse of a cell (after resetting) is not supported. Note that this constraint is not mandatory and is used to simplify the problem at the cost of potentially using more cells for intermediate operations.

- **I/Os of each gate have to be located in the same row and different columns, or alternatively, in the same column and different rows.** Formally,

$$\forall \boldsymbol{gate\ j}\colon \left[\left(\boldsymbol{C_{A_j}} = \boldsymbol{C_{B_j}} = \boldsymbol{C_{E_j}}\right) \cap \left(\boldsymbol{R_{A_j}} \neq \boldsymbol{R_{B_j}} \neq \boldsymbol{R_{E_j}}\right)\right] \tag{4}$$
$$\cup \left[\left(\boldsymbol{C_{A_j}} \neq \boldsymbol{C_{B_j}} \neq \boldsymbol{C_{E_j}}\right) \cap \left(\boldsymbol{R_{A_j}} = \boldsymbol{R_{B_j}} = \boldsymbol{R_{E_j}}\right)\right].$$

- **The simultaneous execution of different NOR gates is possible only when they are aligned in the rows or in the columns**, in order to meet the parallelism requirement described in Section A.

$$\forall \boldsymbol{gates\ j, k}\colon \ \boldsymbol{T_j} \neq \boldsymbol{T_k} \cup \tag{5}$$
$$\left\{\begin{matrix} \left\{\begin{matrix}\left[\left(\boldsymbol{C_{A_j}} = \boldsymbol{C_{A_k}} \cap \boldsymbol{C_{B_j}} = \boldsymbol{C_{B_k}}\right) \cup \left(\boldsymbol{C_{A_j}} = \boldsymbol{C_{B_k}} \cap \boldsymbol{C_{B_j}} = \boldsymbol{C_{A_k}}\right)\right] \\ \cap\, \boldsymbol{C_{E_j}} = \boldsymbol{C_{E_k}}\end{matrix}\right\} \\ \cap \left(\boldsymbol{R_{A_j}} = \boldsymbol{R_{B_j}} = \boldsymbol{R_{E_j}} \cap \boldsymbol{R_{A_k}} = \boldsymbol{R_{B_k}} = \boldsymbol{R_{E_k}}\right) \end{matrix}\right\} \cup$$
$$\left\{\begin{matrix} \left\{\begin{matrix}\left[\left(\boldsymbol{R_{A_j}} = \boldsymbol{R_{A_k}} \cap \boldsymbol{R_{B_j}} = \boldsymbol{R_{B_k}}\right) \cup \left(\boldsymbol{R_{A_j}} = \boldsymbol{R_{B_k}} \cap \boldsymbol{R_{B_j}} = \boldsymbol{R_{A_k}}\right)\right] \\ \cap\, \boldsymbol{R_{E_j}} = \boldsymbol{R_{E_k}}\end{matrix}\right\} \\ \cap \left(\boldsymbol{C_{A_j}} = \boldsymbol{C_{B_j}} = \boldsymbol{C_{E_j}} \cap \boldsymbol{C_{A_k}} = \boldsymbol{C_{B_k}} = \boldsymbol{C_{E_k}}\right) \end{matrix}\right\}.$$

2) Connectivity constraints:

- **Every output of gate *h* that is connected to an input of gate *j* has to be mapped to the same memory cell**, and **the execution of gate *j* has to be performed only after the execution of gate *h*.**

$$\forall \boldsymbol{E_h}, \boldsymbol{x_j} \boldsymbol{\epsilon} \{\boldsymbol{A_j}, \boldsymbol{B_j}\}\ \textit{that are connected}\colon \tag{6}$$
$$\left[\left(\boldsymbol{C_{E_h}} = \boldsymbol{C_{x_j}}\right) \cap \left(\boldsymbol{R_{E_h}} = \boldsymbol{R_{x_j}}\right)\right] \cap \left(\boldsymbol{T_h} < \boldsymbol{T_j}\right).$$

  This configuration does not support movement of data within the memory array. When copy operations are allowed, the mapping can be done to different memory cells, but the order of execution must be maintained.

3) Timing constraints:

- **The execution time of each gate is positive.**

$$\forall \boldsymbol{gate\ j}\colon \boldsymbol{T_j} > \boldsymbol{0}. \tag{7}$$

A toy example of a single-bit full adder optimization is illustrated in Figure 6. The obtained latency is only 10 clock cycles, while naïve implementation of serial execution based on the given ABC netlist has a latency of 15 clock cycles, and previous work that manually optimizes the netlist [4] reached a latency of 13 clock cycles.

V. EXPERIMENTAL RESULTS

To evaluate the SIMPLE synthesis tool, we modified the input library file of the ABC system [25] to convert a given Boolean function into a NOT and two-input NOR netlist[1]. The netlist was mapped and scheduled for minimum latency using the optimization algorithm implemented in the AMPL language [27] and in the Cplex solver [28]. The output of the optimizer gives a general mapping of all input, output and intermediate memristors, and a sequence of operations. We ran the synthesis tool on the LGsynth91 benchmark suite [24] and compared the results with a naïve approach, where the gates are executed serially on the same row in the memory array (thus the number of gates equals the number of clock cycles) before and after the ABC optimization. In addition, we compared the proposed approach with previous work by Chakraborti *et al.* [7], where IMPLY operations [17] were optimized using the Binary Decision Diagram (BDD). The resistors used in each row for IMPLY operations would have resistance values similar to those of the memristors (same order of magnitude). Therefore, when using memristors with similar features for both IMPLY and MAGIC operations, the memory frequency would be equivalent. This allows for a fair comparison between the number of operations required by the SIMPLE synthesis tool, which optimizes the execution of in-memory functions by means of MAGIC operations, and by the method of Chakraborti *et al.*, which is based on IMPLY operations.

Table I lists the synthesis results for a number of combinational benchmarks. The table lists the number of inputs

[1] The SIMPLE MAGIC modified ABC tool can be found at: https://github.com/RotemBenHur/SIMPLE-MAGIC

(In), outputs (Out), and the original number of gates prior to the ABC optimization (Gates), followed by the number of NOR and NOT gates after the ABC optimization (ABC Gates), which is equivalent to the number of execution steps in the naïve approach. The results of the SIMPLE synthesis tool include the area (number of memristors that occupy the memory array for the solution), number of active memristors (Mem), and the latency (OP, number of steps required to compute the function).

The ABC optimization lowers the number of gates on average by 31%. SIMPLE synthesis flow gains another 48% latency improvement on average to a total 89% latency improvement as compared to the original netlist.

Finally, the table shows that the original netlist and the netlist after the ABC optimization have, respectively, an average improvement of 8% and 30%, as compared to Chakraborti *et al.* Hence, using a NOR and NOT netlist offers an advantage over the BDD implementation with IMPLY even without any optimization. SIMPLE improves performance by 94% on average as compared to Chakraborti *et al.*, at the cost of a 44% increase in the number of memristors. Such significant improvement is due to SIMPLE's ability to exploit the parallelism offered by the stateful logic technique, thus revealing the optimal mapping, and by the ability of MAGIC to reduce the number of execution steps as compared to IMPLY. Furthermore, since Chakraborti *et al.* [7] did not consider the number of cycles required for copy operations, their evaluations are optimistic. Hence, the improvement of this work is even more significant.

VI. Conclusions

This paper presents an automatic logic synthesis flow called SIMPLE for optimizing the performance of in-memory computations based on the MAGIC family. SIMPLE solves an optimization problem to minimize the latency of a desired computation. Although solving the optimization problem is mathematically cumbersome and, as a result, computationally intensive, it is solved only once during the design of the mMPU controller, and only the precise mapping is performed in real-time, according to the temporary state of the memory.

Our experimental results show that SIMPLE yields an average performance improvement of 89% as compared to the NOT and NOR netlist prior to optimization, 48% on average as compared to the ABC optimized netlist, and 94% on average as compared to a previous proposed optimization flow.

In future work, we plan to comprehensively evaluate SIMPLE with additional benchmarks and consider removing the non-mandatory constraints (at the cost of an even more computationally intensive optimization problem). Furthermore, we intend to expand SIMPLE to support 3 and 4 input NOR gates, which may also be executed in-memory as a MAGIC operation. Preliminary results show a substantial decrease in the number of gates when including 3-input NOR gates for large benchmarks, probably allowing further latency improvement. Additionally, we plan to extend the optimization problem to other cost functions, such as energy and area, and to develop a real-time address mapping module to complete the synthesis flow.

Acknowledgment

This research was partially supported by Intel Collaborative Research Institute for Computational Intelligence (ICRI-CI), by the Viterbi Fellowship in the Technion Computer Engineering Center, and by EU COST Action IC1401.

References

[1] S. Kvatinsky, E. G. Friedman, A. Kolodny, and U. C. Weiser, "The Desired Memristor for Circuit Designers," *IEEE Circuits and Systems*, vol. 13, no. 2, pp. 17–22, Jun. 2013.

[2] C. Xu, X. Dong, N. P. Jouppi, and Y. Xie, "Design implications of memristor-based RRAM cross-point structures," in *2011 Design, Automation & Test in Europe*, 2011, pp. 1–6.

[3] S. Kvatinsky, D. Belousov, S. Liman, G. Satat, N. Wald, E. G. Friedman, A. Kolodny, and U. C. Weiser, "MAGIC—Memristor-Aided Logic," *IEEE Transactions on Circuits and Systems II: Express Briefs*, vol. 61, no. 11, pp. 895–899, Nov. 2014.

[4] N. Talati, S. Gupta, P. Mane, and S. Kvatinsky, "Logic Design within Memristive Memories Using Memristor-Aided loGIC (MAGIC)," *IEEE Transactions on Nanotechnology*, vol. 15, no. 4, pp. 635–650, Jul. 2016.

[5] R. Ben Hur and S. Kvatinsky, "Memory Processing Unit for In-Memory Processing," in *Proceedings of the IEEE/ACM International Symposium on Nanoscale Architectures*, 2016, pp. 171–172.

[6] R. Ben Hur, N. Talati, and S. Kvatinsky, "Algorithmic Considerations in Memristive Memory Processing Units (MPU)," in *Proceedings of the International Workshop on Cellular Nanoscale Networks and their Applications*, 2016, pp. 1–5.

[7] S. Chakraborti, P. V. Chowdhary, K. Datta, and I. Sengupta, "BDD based synthesis of Boolean functions using memristors," in *Proceedings of the International Design and Test Symposium*, 2014, pp. 136–141.

[8] F. Lalchhandama, B. G. Sapui, and K. Datta, "An Improved Approach for the Synthesis of Boolean Functions Using Memristor Based IMPLY and INVERSE-IMPLY Gates," in *Proceedings of the IEEE Computer Society Annual Symposium on VLSI*, 2016, pp. 319–324.

[9] A. Chakraborty, R. Das, C. Bandopadhyay, and H. Rahaman, "BDD based Synthesis Technique for Design of High-Speed Memristor based Circuits," in *Proceedings of the International Symposium on VLSI Design and Test*, 2016, pp. 1–6.

[10] M. Traiola, M. Barbareschi, A. Mazzeo, and A. Bosio, "XbarGen: a Memristor Based Boolean Logic Synthesis tool," in *Proceedings of the IFIP/IEEE International Conference on Very Large Scale Integration (VLSI-SoC)*, 2016, pp. 1–6.

[11] L. Xie, H. A. Du Nguyen, M. Taouil, K. Bertels, and S. Hamdioui, "Fast boolean logic mapped on memristor crossbar," in *Proceedings of the IEEE International Conference on Computer Design*, 2015, pp. 335–342.

[12] S. Shirinzadeh, M. Soeken, P.-E. Gaillardon, and R. Drechsler, "Fast logic synthesis for RRAM-based in-memory computing using Majority-Inverter Graphs," in *Proceedings of the Design, Automation, and Testing in Europe*, 2016, pp. 948–953.

[13] D. Bhattacharjee and A. Chattopadhyay, "Delay-optimal technology mapping for in-memory computing using ReRAM devices," in *Proceedings of the 35th International Conference on Computer-Aided Design*, 2016, pp. 1–6.

[14] M. Soeken, S. Shirinzadeh, P.-E. Gaillardon, L. G. Amarú, R. Drechsler, and G. De Micheli, "An MIG-based compiler for programmable logic-in-memory architectures," in *Proceedings of the 53rd Annual Design Automation Conference (DAC)*, 2016, pp. 1–6.

[15] L. Amarú, P.-E. Gaillardon, and G. De Micheli, “Majority-Inverter Graph: A novel data-structure and algorithms for efficient logic optimization,” in *Proceedings of the The 51st Annual Design Automation Conference on Design Automation Conference*, 2014, pp. 1–6.

[16] S. Kvatinsky, A. Kolodny, U. C. Weiser, and E. G. Friedman, “Memristor-based IMPLY logic design procedure,” in *Proceedings of the IEEE 29th International Conference on Computer Design*, 2011, pp. 142–147.

[17] S. Kvatinsky, G. Satat, N. Wald, E. G. Friedman, A. Kolodny, and U. C. Weiser, “Memristor-Based Material Implication (IMPLY) Logic: Design Principles and Methodologies,” *IEEE Transactions on Very Large Scale Integration (TVLSI)*, vol. 22, no. 10, pp. 2054–2066, Oct. 2014.

[18] J. H. Poikonen, E. Lehtonen, and M. Laiho, “On Synthesis of Boolean Expressions for Memristive Devices Using Sequential Implication Logic,” *IEEE Transactions on Computer-Aided Design*, vol. 31, no. 7, pp. 1129–1134, Jul. 2012.

[19] P. Teodorovic, S. Dautovic, and V. Malbasa, “Recursive Boolean Formula Minimization Algorithms for Implication Logic,” *IEEE Transactions on Computer-Aided Design*, vol. 32, no. 11, pp. 1829–1833, Nov. 2013.

[20] A. Raghuvanshi and M. Perkowski, “Logic synthesis and a generalized notation for memristor-realized material implication gates,” in *Proceedings of the IEEE/ACM International Conference on Computer- Aided Design*, 2014, pp. 470–477.

[21] F. S. Marranghello, V. Callegaro, M. G. A. Martins, A. I. Reis, and R. P. Ribas, “Factored Forms for Memristive Material Implication Stateful Logic,” *IEEE Journal on Ememrging and Selected Topics in Circuits and Systems*, vol. 5, no. 2, pp. 267–278, Jun. 2015.

[22] F. S. Marranghello, V. Callegaro, A. I. Reis, and R. P. Ribas, “SOP based logic synthesis for memristive IMPLY stateful logic,” in *IEEE International Conference on Computer Design*, 2015, pp. 228–235.

[23] A. Chattopadhyay and Z. Rakosi, “Combinational Logic Synthesis for Material Implication,” in *IEEE/IFIP 19th International Conference on VLSI and System-on-Chip*, 2011, pp. 200–203.

[24] S. Yang, “Logic synthesis and optimization benchmarks user guide: Version 3.0.,” in *MCNC*, 1991.

[25] A. Mishchenko, “ABC: A System for Sequential Synthesis and Verification,” *Berkeley Logic Synthesis and Verification Group, http://www.eecs.berkeley.edu/~alanmi/abc/*. 2012.

[26] R. Ben Hur and S. Kvatinsky, “Memristive Memory Processing Unit (MPU) Controller for In-Memory Processing,” in *Proceedings of the IEEE International Conference on Science of Electrical Engineering*, 2016, pp. 1–5.

[27] R. Fourer, D. M. Gay, and B. W. Kernighan, *AMPL: A Modeling Language for Mathematical Programming*. Duxbury Press/Wadsworth, 1993.

[28] I. IBM ILOG CPLEX Optimization Studio V12.3, “Using the CPLEXR Callable Library and CPLEX Barrier and Mixed Integer Solver Options.” 2011.