Logic Synthesis for RRAM-Based In-Memory Computing

Saeideh Shirinzadeh, *Student Member, IEEE*, Mathias Soeken, *Member, IEEE*, Pierre-Emmanuel Gaillardon, *Senior Member, IEEE*, and Rolf Drechsler, *Fellow, IEEE*

***Abstract*—Design of nonvolatile in-memory computing devices has attracted high attention to resistive random access memories (RRAMs). We present a comprehensive approach for the synthesis of resistive in-memory computing circuits using binary decision diagrams, and-inverter graphs, and the recently proposed majority-inverter graphs for logic representation and manipulation. The proposed approach allows to perform parallel computing on a multirow crossbar architecture for the logic representations of the given Boolean functions throughout a level-by-level implementation methodology. It also provides alternative implementations utilizing two different logic operations for each representation, and optimizes them with respect to the number of RRAM devices and operations, addressing area, and delay, respectively. Experiments show that upper bounds of the aforementioned cost metrics for the implementations obtained by our synthesis approach are considerably improved in comparison with the corresponding existing methods in both area and especially latency.**

***Index Terms*—BDD, in-memory computing, logic synthesis, RRAM.**

I. Introduction

THE ABRUPT switching capability of an oxide insulator sandwiched by two metal electrodes was known from 1960s, but it did not come into interest for several decades until feasible device structures were proposed. Nowadays, a variety of two-terminal devices based on resistance switching property exist which use different materials. These devices possess resistive switching characteristics between two high and low resistance values and are known by various acronyms, such as OxRAM, ReRAM, and resistive random access memory (RRAM) [1]. RRAM devices have also attracted high attention to the theory of memristors proposed in 1971 [2] due to possessing the same resistive characteristics [3].

High scalability of RRAMs [1] makes it possible to implement ultra dense resistive memory arrays [4]. Such architectures using memristive devices are of high interest for their possible applications in nonvolatile memory design [5], [6], digital and analog programmable systems [7]–[9], and neuromorphic computing structures [10].

In [11], it was shown that material implication (IMP) can be used for logic synthesis with resistive devices. In the same work a memristive NAND gate was proposed which enables to realize any Boolean function. This allows advanced computer architectures different from classical von Neumannn architectures by providing memories capable of computing [12], [13]. Although RRAM-based implication logic is sufficient to express any Boolean function, the number of required computational steps to synthesize a given function is a real drawback [14], and only has been addressed by a few works [13], [15].

So far, various memristive logic circuits based on IMP operators have been proposed. An RRAM-based 2-to-1 multiplexer (MUX) containing six RRAM devices was proposed in [16] that requires seven IMP operations. In [17], a similar structure but more efficient in the number of RRAM devices and operations was used for synthesis of Boolean functions based on binary decision diagrams (BDDs). Besides BDDs, and-inverter graphs (AIGs) have been also used for logic synthesis with resistive memories [18]. However, none of these works optimize the utilized data structures with respect to the cost metrics of in-memory computing circuit design.

A novel homogeneous logic representation structure, majority-inverter graph (MIG) was proposed in [19] that uses the majority function together with negation as the only logic operations. MIGs allow higher speeds in design of logic circuits and field-programmable gate array implementations [20]. In comparison with the well-known data structures BDDs and AIGs, MIGs have experimentally shown better results in logic optimization, especially in propagating delay [19]. In particular, MIGs are highly qualified for logic synthesis of RRAM-based circuits since they can efficiently execute the built-in resistive majority operation in RRAM devices [12].

In this paper, we present a comprehensive approach for logic synthesis of RRAM-based in-memory computing circuits using the three mentioned data structures for efficient representation, i.e., BDDs, AIGs, and MIGs. The presented approach includes the following contributions.

1) We present two realizations for each data structure primitives: a) a realization based on IMP and b) a realization that exploits the built-in resistive majority property of RRAM devices [12] denoted by built-in majority operation (MAJ).

Manuscript received March 10, 2017; revised June 30, 2017; accepted August 17, 2017. Date of publication September 7, 2017; date of current version June 18, 2018. This work was supported in part by the University of Bremen's graduate school SyDe through the German Excellence Initiative, CyberCare, under Grant H2020-ERC-2014-ADG 669354, in part by the Swiss National Science Foundation Projects 200021 146600 and 200021 169084, and in part by United States-Israel Binational Science Foundation under Grant 2016016. This paper was recommended by Associate Editor T. Mitra. *(Corresponding author: Saeideh Shirinzadeh.)*

S. Shirinzadeh is with the Department of Mathematics and Computer Science, University of Bremen, 28359 Bremen, Germany (e-mail: s.shirinzadeh@cs.uni-bremen.de).

M. Soeken is with the Integrated Systems Laboratory, EPFL, 1015 Lausanne, Switzerland.

P.-E. Gaillardon is with the Electrical and Computer Engineering Department, University of Utah, Salt Lake City, UT 84112 USA.

R. Drechsler is with the Cyber-Physical Systems, DFKI GmbH, 28359 Bremen, Germany, and also with the Department of Mathematics and Computer Science, University of Bremen, 28359 Bremen, Germany.

Color versions of one or more of the figures in this paper are available online at http://ieeexplore.ieee.org.

Digital Object Identifier 10.1109/TCAD.2017.2750064

2) For each logic representation, we present optimization algorithms with respect to the number of RRAM devices or operations. For BDDs and MIGs, we also propose multiobjective optimization algorithms to lower both cost metrics of in-memory computing addressing the area and delay of the resulting implementations. Experiments confirm the efficiency of the proposed optimization algorithms in comparison with existing approaches.
3) We present efficient design methodologies which enable a certain amount of parallel computing on a multirow multicolumn crossbar, according to the features of the optimized logic representations. We show that the suggested approach guarantees the validity of computations and avoids data distortion during parallel computing requiring only small crossbar dimensions for any Boolean function.
4) We provide a range of design preferences regarding area and latency for logic-in-memory computing, by surveying the three data structures widely used for logic synthesis over two basic operations, i.e., IMP and MAJ.

The remainder of this paper is organized as follows. Section II introduces the employed logic representations, the logic operations for in-memory computing design, and discusses the related work. In Section III, we present our synthesis method for RRAM-based in-memory computing design and the experimental results separately for BDDs, AIGs, and MIGs. Section IV makes a comparison between the exploited logic representations based on their effect on the metrics designating area and latency. Considerations for a crossbar implementation are discussed in Section V, and this paper is concluded in Section VI.

II. Background

A. Logic Representations

1) Binary Decision Diagrams: A BDD (e.g., [21]) is a graph-based representation of a function that is derived from the Shannon decomposition $f = x_i f_{x_i} \oplus \bar{x}_i f_{\bar{x}_i}$. Applying this decomposition recursively allows dividing the function into many smaller subfunctions, which constitute the nodes of BDD representation. By use of complement attribute, a subfunction and its complement can be represented by the same node. An example of a BDD representing a function with four variables is shown in Fig. 1. The nodes corresponding to each input variable x_i represent a BDD level i which needs to be calculated in order, starting from the bottom of the graph to the root node f. Each node at level i has two high and low successors denoted by solid and dashed lines, referring to assignments $x_i = 1$, and $x_i = 0$, respectively. The complemented edges are shown by dots on the successors.

BDDs make use of the fact that for many functions of practical interest, smaller subfunctions occur repeatedly and need to be represented only once. Combined with an efficient recursive algorithm that makes use of caching techniques and hash tables to implement elementary operations, BDDs are a powerful data structure for many applications. BDDs are ordered in the sense that the Shannon decomposition is applied with respect to some given variable ordering which also has an effect on the BDD's number of nodes. Improving the variable ordering for BDDs is NP-complete [22] and many heuristics have been presented that aim at finding a good ordering. Throughout this paper, we consider initial BDD representations before optimization with a fixed ascending variable ordering $x_1 < x_2 < \cdots < x_n$, where n is the number of input variables, e.g., in Fig. 1 $n = 4$ and therefore the ordering is $x_1 < x_2 < x_3 < x_4$.

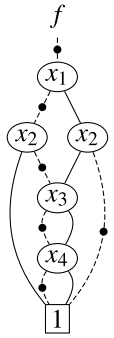


Fig. 1. Initial BDD representation for the function $f = (x_1 \oplus x_2) \vee (x_3 \oplus x_4)$, using the ascending variable ordering and complemented edges.

2) Homogeneous Logic Representations for Circuits: In this paper, we use AIGs [23] and MIGs [19] as homogeneous logic representation. Each node in the graphs represents one logic operation, $x \cdot y$ (conjunction) in case of AIGs, and $M(x, y, z) = x \cdot y + x \cdot z + y \cdot z$ (majority of three) in case of MIGs. Inverters are represented in terms of complemented edges; regular edges represent noncomplemented inputs. Homogeneous logic representations allow for efficient and simpler algorithms due to their regular structure—no case distinction is required for the logic operations. Consequently, such logic representations are the major data structure in state-of-the-art logic synthesis tools.

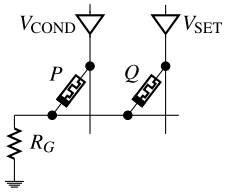
Logic represents in MIGs are at least as compact as in AIGs, since each AND node can be mapped to exactly one majority node; we have $x \cdot y = M(x, y, 0)$. However, even smaller MIGs can be obtained if their capability of compactness is fully exploited such that no node in the graph has constant inputs [24].

A Boolean algebra was proposed in [19] in order to optimize MIGs. The following set (Ω) includes the primitive transformations that can be applied to an existing MIG to get a more efficient representation

$$\Omega \begin{cases} \textbf{Commutativity} - \Omega.C \\ M(x, y, z) = M(y, x, z) = M(z, y, x) \\ \textbf{Majority} - \Omega.M \\ M(x, x, z) = x \qquad M(x, \bar{x}, z) = z \\ \textbf{Associativity} - \Omega.A \\ M(x, u, , M(y, u, z)) = M(z, u, M(y, u, x)) \\ \textbf{Distributivity} - \Omega.D \\ M(x, y, M(u, v, z)) = M(M(x, y, u), M(x, y, v), z) \\ \textbf{InverterPropagation} - \Omega.I \\ \overline{M}(x, y, z) = M(\bar{x}, \bar{y}, \bar{z}). \end{cases}$$

It was proven in [24] that any MIG can be transformed to another logically equivalent MIG using only Ω axioms. It means that reaching a desired MIG optimized with respect to the considered cost metric is possible by applying Ω, however, the length of transformation sequence might be impractical. To solve this problem, a more advanced set of transformations derived from the basic rules in Ω was proposed in [19] which was denoted by Ψ. We only refer to *complementary associativity* ($\Psi.C$) from the set Ψ that is used in this paper, and is formally expressed by

$$\Psi.C : M(x, u, M(y, \bar{u}, z)) = M(x, u, M(y, x, z)).$$

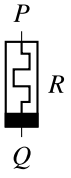


(a)

p	q	q'
0	0	1
0	1	1
1	0	0
1	1	1

(b)

Fig. 2. IMP operation. (a) Implementation of IMP using RRAM devices. (b) Truth table for IMP ($q' \leftarrow p$ IMP $q = \bar{p} + q$) [11].



P	Q	R	R'
0	0	0	0
0	1	0	0
1	0	0	1
1	1	0	0

$R' = P \cdot \overline{Q}$

P	Q	R	R'
0	0	1	1
0	1	1	0
1	0	1	1
1	1	1	1

$R' = P + \overline{Q}$

Fig. 3. Intrinsic majority operation within an RRAM device.

B. Logic Operations for RRAM-Based Design

1) Material Implication: IMP and FALSE operation, i.e., assigning the output to logic 0, are sufficient to express any Boolean function [11]. Fig. 2 shows the implementation of an IMP gate which was proposed in [11]. P and Q designate two resistive devices connected to a load resistor R_G. Three voltage levels V_{SET}, V_{COND}, and V_{CLEAR} are applied to the devices to execute IMP and FALSE operations by switching between low-resistance (logic 1) or high-resistance (logic 0) states.

The FALSE operation can be performed by applying V_{CLEAR} to an RRAM device. An RRAM device can be also switched to logic 1 by applying a voltage larger than a threshold V_{SET} to its voltage driver. To execute IMP, two voltage levels V_{SET} and V_{COND} are applied to the switches P and Q simultaneously. The magnitude of V_{COND} is smaller than the required threshold to change the state of the switch. However, the interaction of V_{SET} and V_{COND} can execute IMP according to the current states of the switches, such that switch Q is set to 1 if $p = 0$ and it retains its current state if $p = 1$ [11].

2) Built-in Majority Operation: RRAM devices have two terminals and their internal resistance R can be switched between two logic states 0 and 1 designating high and low resistance states, respectively. Denoting the top and bottom terminals by P and Q, the memory can be switched with a negative or positive voltage V_{PQ} based on the device polarity. Here, we assume that the logic statements ($P = 1, Q = 0$) switches the RRAM device to logic 1, ($P = 0, Q = 1$) switches the device to logic 0, and ($P = Q$) does not change the current state of the device. Accordingly, we can make the truth tables shown in Fig. 3 for the next sate of the switch (R') when the current state (R) is either 0 or 1. In the following, the Boolean relations represented by tables in Fig. 3 are extended which formally express the MAJ of RRAM devices [12]:

$$\begin{aligned} R' &= (P \cdot \overline{Q}) \cdot \overline{R} + (P + \overline{Q}) \cdot R \\ &= P \cdot R + \overline{Q} \cdot R + P \cdot \overline{Q} \cdot \overline{R} \\ &= P \cdot R + \overline{Q} \cdot R + P \cdot \overline{Q} \cdot R + P \cdot \overline{Q} \cdot \overline{R} \\ &= P \cdot R + \overline{Q} \cdot R + P \cdot \overline{Q} \\ &= M(P, \overline{Q}, R). \end{aligned}$$

The operation above is referred to 3-input resistive majority $RM_3(x, y, z)$, such as $RM_3(x, y, z) = M(x, \bar{y}, z)$ [12]. According to RM_3, the next state of a resistive switch is equal to a result of a built-in majority gate when one of the three variables x, y, and z is already preloaded and the variable corresponding to the logic state of the bottom electrode is inverted. We denote this intrinsic property of RRAM devices by MAJ which provides an alternative for IMP and enables shorter computational length for synthesis of Boolean functions using resistive switches.

C. Related Work

So far, few synthesis approaches using logic representations have been proposed for in-memory computing. All the existing approaches in this area exploit IMP for realization of the nodes of their employed graph-based data structures. The unfavorable nature of sequential operations for RRAM-based in-memory computing has been mostly exploited to reduce the number of required RRAM devices. Some approaches evaluate the graph-based representation completely in sequence such that only a single node can be computed each time [18], [25]. These evaluation methods increase the length of computational sequences in comparison with the parallel evaluation proposed in [17] in which nodes of equal latency are computed at the same time at a higher cost in area. However, the approach presented in [18] tries to avoid higher increase in the number of operations by providing a tradeoff between the additional number of operations and RRAM devices required for maintaining the intermediate results.

In [25], IMP was used to synthesize combinational logic circuits with resistive memories using or-inverter graphs (OIGs). The approach applies an extension of the delay minimization algorithm proposed in [26] to the OIGs and also uses an area minimization to lower the costs of the equivalent circuits constructed with resistive memories. Synthesis of in-memory computing circuits using OIGs can be also possible with NOR gates based on memristor-aided loGIC (MAGIC) proposed in [27]. MAGIC provides a memristive stateful logic, which has experimentally shown lower latency in comparison with IMP [15].

Another approach using AIGs was proposed in [18] for synthesis of in-memory computing logic circuits. The approach uses the state-of-the-art synthesis tool ABC [28] to map an arbitrary Boolean function to an AIG and optimize it. An AIG representing a given function is then mapped to an equivalent network of IMP gates (Fig. 2) according to the IMP-based realization of NAND gate proposed in [11]. The approach executes a given Boolean function using $N + 2$ RRAM devices, where N is the number of *input RRAM devices*, which keep their initial values until the target function is executed and 2 is the number of *work RRAM devices*, which states are changed during the operations by intermediate results or the final output. Nevertheless, some extra RRAM devices are also considered to maintain values of the IMP gates which have more than one fanout.

BDD-based synthesis of Boolean functions using resistive memories has been proposed in [17]. Two IMP-based realizations are proposed for a 2-to-1 MUX one for a minimum number of resistive switches and the other for a minimum number of operations when lower latency is of higher importance than area. It has not been referred to any BDD optimization method in [17] to lower either the number of RRAM devices or operations. For a given Boolean function,

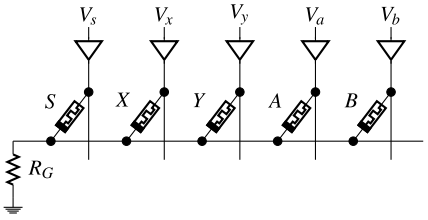


Fig. 4. Realization of an IMP-based MUX using RRAM devices [17].

the approach maps the corresponding BDD representation to a netlist of RRAM devices using any of the MUX realizations. This is carried out using two mapping approaches, one fully sequential which is slow but needs a small number of RRAM devices, and the other partially parallel which performs much faster but needs some considerations for the complemented edges and fanouts.

III. RRAM-Based In-Memory Computing Design

In this section, we present our proposed synthesis approach for RRAM-based logic-in-memory computing using the three representations explained before. For each representation, two realizations for a graph node are presented using IMP and MAJ as well as the methodology to map the graph to its equivalent circuit constructed by RRAM devices.

We present optimization algorithms for each logic representation to lower the cost metrics of the resulting in-memory computing circuits dissimilar to the conventional optimization algorithms that are mainly designed to reduce the size, i.e., the number of nodes of the graph also called area, or depth, i.e., the number of levels of the graph.

In the rest of this section, we present the node realizations, design methodology, optimization, and the experimental results for each graph-based representations introduced in Section II-A in order.

A. BDD-Based Synthesis for In-Memory Computing Design

1) Realization of Multiplexer Using RRAM Devices: Fig. 4 shows the IMP-based realization for 2-to-1 MUX proposed in [17]. The implementation requires six operations and five RRAM devices of which three, named S, X, and Y, store the inputs, and the two others, A and B, are required for operations. The corresponding implication steps of the MUX realization shown in Fig. 4 are as follows.

1) $S = s, X = x, Y = y, A = 0, B = 0$.
2) $a \leftarrow s$ IMP $a = \bar{s}$.
3) $a \leftarrow y$ IMP $a = \bar{y} + \bar{s}$.
4) $b \leftarrow a$ IMP $b = y \cdot s$.
5) $s \leftarrow x$ IMP $s = \bar{x} + s$.
6) $b \leftarrow s$ IMP $b = x \cdot \bar{s} + y \cdot s$.

In the first step, devices keeping the input variables and the two extra work switches are initialized. The remaining steps are performed by sequential IMP operations that are executed by applying simultaneous voltage pulses V_{COND} and V_{SET}.

To find the MAJ-based realization of MUX, we first express the Boolean function of an MUX with majority gates and then simply convert it to RM_3 by adding a complement attribute to each gate. For this purpose, the AND and OR operations are represented by majority gates using a constant as the third input variable, i.e., 0 for AND and 1 for OR [19]. Accordingly, an MUX with input variables x, y and a select input s can be expressed as

$$\begin{aligned} x \cdot \bar{s} + y \cdot s &= M(M(x, \bar{s}, 0), M(y, s, 0), 1) \\ &= M(M(x, \bar{s}, 0), M(y, 0, s), 1) \\ &= RM_3\big(RM_3(x, s, 0), \overline{RM_3}(y, 1, s), 1\big). \end{aligned}$$

The equations above can be executed by three RM_3 operations as well as a negation. Therefore, the MAJ-based realization of the MUX can be obtained by the following operations after a data loading step.

1) $S = s, X = x, Y = y, A = 0, B = 0, C = 1$.
2) $P_A = x, Q_A = s, R_A = 0 \Rightarrow R'_A = x \cdot \bar{s}$.
3) $P_S = y, Q_S = 1, R_S = s \Rightarrow R'_S = y \cdot s$.
4) $P_B = 1, Q_B = s, R_B = 0 \Rightarrow R'_B = \overline{y \cdot s}$.
5) $P_C = a, Q_C = b, R_C = 1 \Rightarrow R'_C = x \cdot \bar{s} + y \cdot s$.

The proposed MAJ-based MUX can be realized quite similarly to the IMP-based circuit shown in Fig. 4 such that the bottom electrodes of the switches are electrically connected via a horizontal nanowire and the switching can be done by applying the three discussed voltage levels to the top electrodes. As can be seen, the MAJ-based realization of MUX needs one more RRAM devices and one less operation. Considering area and delay two equally important cost metrics, using IMP or MAJ does not make a difference in the circuits synthesized by the proposed BDD-based approach. Indeed, the MAJ-based realization of BDD nodes allows faster circuits, while the IMP-based realization leads to circuits with smaller area consumption. Such property in both realizations can be exploited when higher efficiency in delay or area is intended.

2) Design Methodology for BDD-Based Synthesis: In order to escape heavy delay penalties, we assume parallelization per level for BDD-based synthesis [17], [29]. As explained before, in the parallel implementation, each time one BDD level is evaluated entirely starting from the level designating the last ordered variable to the first ordered variable the so-called root node. This is performed through transferring the computation results between successive levels, i.e., using the outputs of each computed level as the inputs of the next level. Using IMP, the results of previous levels are read and copied, wherever required within the first loading step of the next level, while for executing MAJ the results are read and then applied as voltages to the rows and columns.

Regardless of the possible fanouts and complemented edges in the BDD, the number of RRAM devices required for computing by this approach is equal to five or six times the maximum number of nodes in any BDD level. In a similar way, the number of operations is six or five times the number of BDD levels, for the IMP-based and MAJ-based realizations, respectively. A multiple row crossbar architecture entirely based on resistive switches was proposed in [30], which can be used to realize the presented parallel evaluation.

The cost metrics of the proposed BDD-based synthesis approach are given in Table I. However, the larger part of the costs representing area and delay of the resulting circuits are explained above, some additional RRAM devices addressing complemented edges and fanouts are still required.

Every complemented edge in the BDD requires an NOT gate to invert its logic value. As shown in the computational steps for both IMP-based and MAJ-based realizations, inverting a variable can be executed after an operation with a zero loaded RRAM device (see step 2 in the IMP-based MUX and step 4 in the MAJ-based MUX descriptions). Accordingly, for each MUX with a complemented input an extra RRAM device

TABLE I
COST METRICS OF LOGIC REPRESENTATIONS FOR RRAM-BASED IN-MEMORY COMPUTING

Metric	Definition\Value	
$\mathbf{N}_i$	No. of nodes in the i^{th} level	
$\mathbf{CE}_i$	No. of ingoing complemented edges in the i^{th} level	
$\mathbf{RE}_i$	No. of ingoing regular edges in the i^{th} level	
FO	Maximum no. of nonconsecutive fanouts in any BDD level	
D	The depth of the graph	
$\mathbf{L}_{CE}$	No. of levels with ingoing complemented edges	
$\mathbf{L}_{RE}$	No. of levels with ingoing regular edges	
R	No. of RRAM devics	
OP	No. of operations	
#R	BDD: $\max_{0\le i\le D}(K\cdot N_i+CE_i)+FO$	IMP : $K=5$, MAJ : $K=6$
	AIG: IMP : $\max_{0\le i\le D}(3\cdot N_i+RE_i)$	MAJ : $\max_{0\le i\le D}(3\cdot N_i+CE_i)$
	MIG: $\max_{0\le i\le D}(K\cdot N_i+CE_i)$	IMP : $K=6$, MAJ : $K=4$
#OP	BDD: $K\cdot D+L_{CE}$	IMP : $K=6$, MAJ : $K=5$
	AIG: IMP : $3\cdot D+L_{RE}$	MAJ : $3\cdot D+L_{CE}$
	MIG: $K\cdot D+L_{CE}$	IMP : $K=10$, MAJ : $K=3$

should be considered and set to FALSE ($Z=0$) that can be performed in parallel with the first loading step without any increase in the number of steps. Then, an IMP or MAJ operation should be executed to complete the logic NOT operation. It is obvious that the required operations for all complemented edges in a level can be carried out simultaneously that means for any level with ingoing complemented edges only one extra step is required. This implies that the number of additional steps required for inverting all of the complemented edges cannot exceed the number of BDD levels. Therefore, the number of steps to evaluate a BDD possessing complemented edges is equal to the number of BDD levels with ingoing complemented edges besides the basic value required for the level counts [29].

It is obvious that the RRAM devices keeping the outputs of each BDD level can be reused and assigned to the inputs of the next successive level. Nevertheless, the results of nodes targeting levels which are not right after their origin level might be lost during computations if their corresponding RRAM devices are rewritten by the next operations. Thus, we consider extra RRAM devices for such nonconsecutive fanouts to retain the result of their origin nodes to be used as an input signal of their target nodes. The required number of RRAM devices for this is equal to the maximum number of such fanouts over all BDD levels. This will not increase the number of steps because copying the results of nodes with nonconsecutive fanouts in additional RRAM devices and using the stored value in the fanouts' targets can be performed simultaneously in the first data loading step of nodes on the both sides of the fanouts.

3) BDD Optimization for RRAM-Based Circuit Design: Optimization of BDDs in this paper is carried out as a bi-objective problem aiming at minimizing the number of RRAM devices and computational steps simultaneously, i.e., finding a tradeoff between the number of RRAM devices and operations of the resulting circuits. For this purpose, we have exploited a multi-objective genetic algorithm (MOGA). The general framework of MOGA employed BDD optimization algorithm is based on nondominated sorting genetic algorithm [31] that has been experimentally proven useful for solving NP-complete problems, such as BDD optimization [29], [32]. MOGA is also capable of handling higher priority to any of the cost metrics, which allows to design smaller in-memory computing circuits at a fair cost of latency or vice versa. We refer to [32] for the details of MOGA.

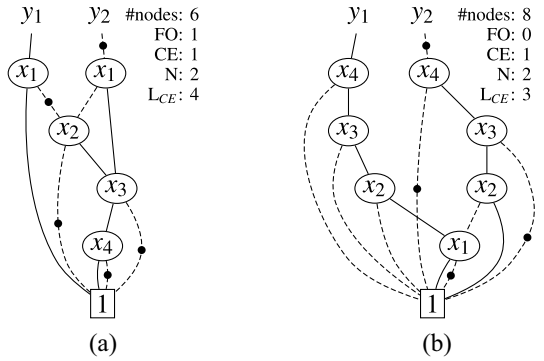


Fig. 5. Cost metrics of RRAM-based in-memory computing for an arbitrary BDD, (a) before (initial), and (b) after optimization (optimized).

Fig. 5 shows an example with two BDDs both representing a 4-variable 2-output Boolean function. The left BDD has the initial ordering, whereas the second BDD has the ordering obtained by MOGA. The number of required RRAM devices for computing BDD levels ($N+CE$) (see Table I) is equal before and after optimization since both BDDs have a maximum number of two nodes and one ingoing complemented edge. However, there is a nonconsecutive fanout of node x_3 targeting x_1 before optimization requiring an extra RRAM device to maintain the intermediate result. In the optimized BDD the inputs of all of the nodes come from the consecutive levels or the constant 1 which has reduced the number of required RRAM devices by 1. The number of operations has been also reduced after optimization since one level has been released from complemented edges.

As can be seen, the numbers of RRAM devices and operations decrease although the number of BDD nodes increases. The effect of BDD optimization sounds to be too small for the example function by reducing each one of the cost metrics only by one. Nevertheless, this reduction can be much more visible for larger functions due to the higher possibility of finding BDDs with smaller number of nonconsecutive fanouts, complemented edges, and level sizes caused by larger search space.

4) Results of BDD-Based Synthesis: We have evaluated our proposed synthesis approaches using a set of 25 benchmark functions selected from LGsynth91 [33]. The results of the BDD-based and AIG-based synthesis approaches have been also compared with the similar existing approaches introduced in Section II-C, which use the same data structures for RRAM-based in-memory computing. For each benchmark function, MOGA has been run ten times with a termination criterion of 500 generations. The population is three times as large as the number of inputs of each function with a maximum allowed size of 120. The rest of the experimental setup including the genetic operators and their probabilities are the same as used in [32].

TABLE II
COMPARISON OF RESULTS BY GENERAL AND PRIORITIZED MOGA WITH CHAKRABORTI *et al.* [17]

		Chakraborti et al. [17]		MOGA IMP		MOGA MAJ		MOGA with priority to *R* IMP		MOGA with priority to *R* MAJ		MOGA with priority to *OP* IMP		MOGA with priority to *OP* MAJ	
Benchmark	PI/PO	*R*	*OP*	*R*	*OP*	*R*	*OP*	*R*	*OP*	*R*	*OP*	*R*	*OP*	*R*	*OP*
5xp1_90	7/10	84	73	57	49	62	42	57	49	62	42	65	42	62	42
alu4_98	14/8	642	334	697	93	833	78	590	96	847	77	1014	77	1069	77
apex1	45/45	1626	705	1139	282	1334	236	1082	283	1312	237	3040	277	3185	232
apex2	39/3	122	237	165	235	186	196	172	237	186	196	188	235	173	196
apex4	9/19	2073	447	2028	63	2558	54	2028	63	2558	54	2224	62	2588	53
apex5	117/88	806	888	537	772	582	654	509	775	582	654	591	771	704	654
apex6	135/99	770	1169	179	814	189	679	160	815	180	679	220	813	235	678
apex7	49/37	290	437	118	328	143	279	102	332	134	280	190	328	151	279
b9	41/21	125	298	64	267	73	226	59	269	79	226	77	267	92	226
clip	9/5	120	89	93	63	107	54	93	63	107	54	93	63	107	54
cm150a	21/1	56	127	28	127	30	106	28	127	30	106	28	127	30	106
cm162a	14/5	46	102	42	91	42	78	32	93	36	79	38	89	43	75
cm163a	16/5	42	116	29	112	31	92	29	112	30	92	31	108	32	92
cordic_138	23/2	32	149	25	140	24	117	21	140	24	117	26	140	29	117
misex1_178	8/7	83	69	79	50	91	42	59	52	69	44	79	50	91	42
misex3_180	14/14	444	185	436	91	503	77	436	91	503	77	681	86	781	72
parity	16/1	23	113	6	112	7	96	6	112	7	96	6	112	7	96
seq_201	41/35	1566	692	1199	250	1320	210	1129	251	1320	210	1207	248	1398	207
t481_208	16/1	26	107	12	100	14	84	12	100	14	84	16	100	30	84
table5	17/15	580	168	667	109	768	92	637	113	763	95	1346	107	1511	90
too_large	38/3	282	232	214	229	174	191	164	232	232	191	182	229	212	191
x1	51/35	230	398	185	333	219	282	172	339	203	289	186	333	217	282
x2	10/7	60	80	45	65	52	55	42	67	49	57	45	65	52	55
x3	135/99	770	1169	221	813	252	678	161	815	191	679	215	813	252	678
x4	94/71	401	642	177	573	324	479	177	573	231	481	209	573	333	479
AVG		451.96	361.04	337.67	246.44	396.72	207.08	318.28	247.96	389.96	207.84	479.88	244.6	535.36	206.28

PI/*PO*: number of primary inputs/number of primary outputs, *R*: number of RRAM devices, *OP*: number of operations

Table II presents the results of the three versions of MOGA and compares them with results of the BDD-based synthesis approach proposed in [17]. For MOGA with priority to the number of RRAM devices and operations, we chose results with the smallest number of RRAM devices and operations among all runs and populations. The results shown in the table for the general MOGA have been also selected such that they represent a good tradeoff between the minimum and maximum values found by the prioritized algorithms. It is worth mentioning that the runtime varies between 0.56 to 187.22 s for the benchmark functions 5xp1_90 and seq_201, respectively.

According to Table II, the number of RRAM devices obtained by MOGA with priority to *R* for the IMP-based realization is reduced by 5.74% on average compared to the corresponding value by the general MOGA. MOGA with priority to the number of operations also achieves smaller latency by reducing the average operation count up to 0.74% in comparison to the general MOGA. It should be noted that optimization cannot noticeably lower the number of operations. As shown in Table I, the main contribution to the operation count is the number of BDD levels, i.e., the number of input variables and hence is not changeable.

In comparison with results of [17] our BDD-based synthesis approach has achieved better performance in both cost metrics. The average values of results over the whole benchmark set by the general MOGA for the IMP-based realization, which is also used by [17], shows reduction of 21.11% and 31.74% in the number of RRAM devices and operations, respectively. The reduction in the number of operations reaches 42.64% for the MAJ-based realization which also has 12.22% smaller number of RRAM devices.

B. AIG-Based Synthesis for In-Memory Computing Design

1) Realization of NAND/AND Gate Using RRAM Devices: Realization of NAND gate using resistive switches based on IMP has been proposed in [11]. The proposed NAND gate in [11] corresponds to a node with complemented fanout in an AIG and therefore can be utilized as the IMP-based implementation realizing AIGs with RRAM devices. In this case, a FALSE operation is required for any regular edge in the graph. The implementation proposed in [11] requires three resistive memories connected by a common horizontal nanowire to a load resistor, i.e., structurally similar to the circuit shown in Fig. 4 with a different number of switches. The interaction of the tri-state voltage drivers on the RRAM devices execute the NAND operation within three computational steps listed below.

1) $X = x, Y = y, A = 0$.
2) $a \leftarrow x \,\text{IMP}\, a = \bar{x}$.
3) $a \leftarrow y \,\text{IMP}\, a = \bar{x} + \bar{y}$.

Using MAJ, AIG can be also implemented with equal number of RRAM devices and operations. A majority operation of two variables x and y together with a constant logic value of 0 ($M(x, 0, y)$) [19] executes the AND operation. This corresponds to MAJ(x, 1, y) which only needs one extra operation to preload operand y in a resistive switch. The required steps are as follows.

1) $X = x, Y = y, A = 0$.
2) $P_A = y, Q_A = 1, R_A = 0 \Rightarrow R'_A = y$.
3) $P_A = x, Q_A = 1, R_A = y \Rightarrow R'_A = x \cdot y$.

2) Design Methodology for AIG-Based Synthesis: Although both of the realizations using IMP and MAJ for the AIG-based synthesis approach impose sequential circuit implementations, they allow a reduction in area by reusing RRAM devices released from previous computations. According to the parallel evaluation method, we only consider one AIG level each time, such that the employed RRAM devices to evaluate the level can be reused for the next levels. Starting from the inputs of the graph, the RRAM devices in a level are released when all the required operations are executed. Then, the RRAM devices are

reused for the upper level and this procedure is continued until the target function is evaluated. Depending on the use of IMP or MAJ in the realization, such an implementation requires as many NAND or AND gates as the maximum number of nodes in any level of the AIG. Hence, the corresponding number of RRAM devices and operations for synthesizing the AIG is three times the number of required majority gates and three times the number of levels, respectively.

However, still some additional RRAM devices should be allocated for the required NOT operations, i.e., the regular edges in the IMP-based realization, where the outputs of AIG nodes are already negated due to being implemented by NAND gates, and the complemented edges in the MAJ-based realization. Table I shows the number of RRAM devices and computational steps of the resulting RRAM-based circuits. Since the implementation starts from the input of AIG, the ingoing regular edges for the IMP-based realization, and the ingoing complemented edges for the MAJ-based realization of any level should be first inverted similarly to the procedure explained for BDDs. Therefore, the total number of RRAM devices required for the synthesis of the whole graph for both IMP-based and MAJ-realizations is equal to the maximum of three times the number of nodes in the level plus the number of ingoing edges to be inverted over all AIG levels.

3) AIG Optimization for RRAM-Based Circuit Design: For AIG optimization we have used ABC [28] commands. To address the area of the resulting circuits using RRAM devices we use the command *dc2* which minimizes the number of nodes in the graph. Latency of the circuits has been also reduced before mapping them to their corresponding netlist of RRAM devices by another ABC command *if -x -g*. The command minimizes the depth of AIG which is actually the most significant term in the required number of operations given in Table I due to the factor of three for both IMP-based and MAJ-based realizations. Both of the area and depth AIG rewriting commands by ABC do not target the extra number of RRAM devices and computational steps caused by the NOT operations for synthesis. Nevertheless, applying any of the aforementioned commands iteratively can noticeably reduce the cost metrics of RRAM-based in-memory computing.

It should be noted that we cannot optimize AIG for both cost metrics since area minimization leads to worsening the latency and on the other hand depth minimization increases the number of nodes in the graph. Thus, according to the application one can choose the optimization command regarding the area or delay of the resulting circuits.

4) Results of AIG-Based Synthesis: Results of the proposed AIG-based synthesis approach for in-memory computing are presented in Table III for both area and depth rewriting methods by ABC [28]. A quick look at Table III reveals that the number of RRAM devices is smaller for the MAJ-based realization, while the operation counts are almost equal. According to Table III, area and depth rewriting reduce the total number of RRAM devices and operations, respectively, by 24.31% and 10.04% on average compared to each other.

Table IV makes a comparison with the AIG-based approach proposed in [18] for a different benchmark set with single output functions, i.e., $PO = 1$. Since the number of required RRAM devices for the benchmark set are not given in [18], we can only compare with respect to the number of operations. The number of operations obtained by our proposed method using the IMP-based realization, which is also used by [18], is seven times smaller than the that of [18] for both rewritings. Furthermore, the method proposed in [18] fails to keep the number of computational steps at a reasonable value when the number of inputs increases. For example, the number of operations by [18] for functions *sym10_d* and *t481_d* is equal to 1172 and 1564, respectively. While using our method, both functions can be synthesized with less than 80 operations.

TABLE III
RESULTS OF AIG-BASED SYNTHESIS USING SIZE AND DEPTH REWRITING BY ABC [28]

	Area minimization				Depth minimization			
	IMP		MAJ		IMP		MAJ	
Benchmark	*R*	*OP*	*R*	*OP*	*R*	*OP*	*R*	*OP*
5xp1_90	60	38	45	39	131	35	133	35
alu4_98	692	78	696	79	1046	71	1010	71
apex1	1513	67	1346	67	2192	55	1816	55
apex2	171	83	189	87	302	70	325	71
apex4	2153	67	2040	67	2710	55	2282	55
apex5	698	51	630	51	1033	55	831	55
apex6	639	54	685	55	791	47	681	47
apex7	147	59	147	58	228	47	196	47
b9	123	31	123	31	128	27	123	27
clip	78	37	82	37	176	39	176	39
cm150a	63	36	72	39	95	29	81	28
cm162a	42	39	42	39	42	31	42	31
cm163a	48	35	48	35	48	27	48	27
cordic_138	71	37	73	39	100	37	108	37
misex1_178	49	31	55	31	66	25	70	27
misex3_180	900	79	820	79	1239	67	1105	67
parity	64	38	64	43	64	59	64	67
seq _201	1040	86	1000	87	1723	71	1453	71
t481_208	48	25	48	27	100	37	92	37
table5	1155	83	869	83	1878	71	1466	71
too_large	153	80	159	83	279	82	321	83
x1	267	39	271	39	382	39	410	39
x2	45	27	35	27	53	27	55	27
x3	573	54	699	55	801	37	711	37
x4	298	37	282	37	371	31	301	31
AVG	443.6	51.6	420.8	52.56	639.12	46.84	556	47.28

It is worth mentioning that the runtime for each benchmark function in both Tables III and IV is in the range of milliseconds.

C. MIG-Based Synthesis for In-Memory Computing Design

1) Realization of Majority Gate Using RRAM Devices: We propose two realizations for majority gate based on IMP and MAJ [34]. The proposed IMP-based realization of majority gate is similar to the circuit shown in Fig. 4 with six of RRAM devices. It also requires ten sequential steps to execute the majority function. The corresponding steps for executing the majority function are as follows.

1) $X = x, Y = y, Z = z, A = 0, B = 0, C = 0$.
2) $a \leftarrow x$ IMP $a = \bar{x}$.
3) $b \leftarrow y$ IMP $b = \bar{y}$.
4) $y \leftarrow a$ IMP $y = x + y$.
5) $b \leftarrow x$ IMP $b = \bar{x} + \bar{y}$.
6) $c \leftarrow y$ IMP $c = \overline{x + y}$.
7) $c \leftarrow z$ IMP $c = \overline{x \cdot z + y \cdot z}$.
8) $a = 0$.
9) $a \leftarrow b$ IMP $a = x \cdot y$.
10) $a \leftarrow c$ IMP $a = x \cdot y + y \cdot z + x \cdot z$.

Three RRAM devices denoted by X, Y, and Z keep input variables and the remaining three other RRAM devices A, B, and C are required for retaining the intermediate results and the final output. In the first step, the input variables are loaded and the other RRAM devices are assigned to FALSE to be

TABLE IV
COMPARISON OF RESULTS BY THE PROPOSED AIG-BASED SYNTHESIS WITH BÜRGER *et al.* [18]

		Bürger et al. [18]	Area minimization				Depth minimization			
			IMP		MAJ		IMP		MAJ	
Benchmark	*PI*	*OP*	*R*	*OP*	*R*	*OP*	*R*	*OP*	*R*	*OP*
9sym_d	9	1418	231	71	225	71	370	62	342	62
con1f1	7	18	18	18	18	19	18	21	18	21
con2f2	7	19	15	22	15	23	23	19	25	18
exam1_d	3	12	9	17	9	19	9	17	9	19
exam3_d	4	12	12	21	12	23	19	18	21	18
max46_d	9	427	138	61	134	63	195	49	177	51
newill_d	8	50	24	33	24	34	59	41	61	42
newtag_d	8	21	24	23	24	22	24	27	24	27
rd53f1	5	27	16	21	16	23	26	26	22	27
rd53f2	5	57	23	31	25	31	44	26	44	27
rd53f3	5	32	16	24	16	27	16	24	16	27
rd73f1	7	238	79	47	81	47	127	42	153	43
rd73f2	7	46	24	24	24	27	24	31	24	35
rd73f3	7	104	29	34	27	35	66	38	62	38
rd84f1	8	351	128	54	128	55	167	51	153	51
rd84f2	8	47	32	24	32	27	32	31	32	35
rd84f3	8	23	24	15	24	12	24	19	24	15
rd84f4	8	345	127	51	137	50	186	51	182	50
sao2f1	10	102	31	48	30	50	68	42	68	43
sao2f2	10	112	63	34	65	35	103	42	97	43
sao2f3	10	380	63	54	62	54	108	54	100	55
sao2f4	10	252	66	53	62	54	126	51	122	51
sym10_d	10	1172	375	79	401	79	575	66	569	67
t481_d	16	1564	277	93	275	95	500	78	452	79
xor5	5	32	16	24	16	27	16	24	16	27
AVG		274.44	74.2	39.04	75.28	40.08	117	38	112.52	38.84

used later for the next operations. Another FALSE operation is also performed in step 8, to clear an RRAM device which is not required anymore for inverting an intermediate result which is not required anymore. Finally, the Boolean function representing a majority gate is executed by implying results from the seventh and ninth step.

It is obvious that the MAJ-based majority gate can be realized with smaller number of RRAM devices and computational steps due to benefiting from the discussed built-in majority property. Using MAJ, the majority gate will require only four RRAM devices that can be placed in the same structure shown in Fig. 4. Furthermore, the majority function can be executed within only three steps carrying out simple operations. The MAJ-based computational steps for the proposed RRAM-based realization are as follows.

1) $X = x, Y = y, Z = z, A = 0$.
2) $P_A = 1, Q_A = y, R_A = 0 \Rightarrow R'_A = \bar{y}$.
3) $P_Z = x, Q_Z = \bar{y}, R_Z = z \Rightarrow R'_Z = M(x, y, z)$.

In the first step, the initial values of input variables as well as an additional RRAM device are loaded by applying V_{SET} or V_{CLEAR} to their voltage divers. Step 2 executes the required NOT operation in RRAM device A. This can be done with applying appropriate voltage levels V_{SET} or V_{COND} to switch A, for cases $y = 0$ and $y = 1$, respectively. In the last step, the majority function is executed by use of MAJ at RRAM device Z by applying any of the three voltage levels corresponding the difference between logic states of x and $\bar{y}$.

2) Design Methodology for MIG-Based Synthesis: The number of RRAM devices and operations for the proposed MIG-based synthesis approach are given in Table I. The method of mapping MIGs to equivalent RRAM-based in-memory computing circuits is exactly similar to the design methodology for AIGs with MAJ-based realization. Since both IMP-based and MAJ-based realizations proposed for MIGs represent majority gate without an extra negation, the same formula can be used for them with different constant factors addressing the number of RRAM devices and operations required by each realization [34].

3) MIG Optimization for RRAM-Based Circuit Design: In general, MIG optimization is performed by applying a set of valid transformations to an existing MIG to find an equivalent MIG that is more efficient with respect to the considered cost metrics. In this section, we present the three MIG optimization algorithms tackling the cost metrics of logic synthesis with RRAM devices. The first proposed algorithm considers both cost metrics simultaneously [34], while the others aim at reducing the number of operations [34] or RRAM devices [35].

In [19], two algorithms for conventional MIG optimization in terms of delay and area have been proposed, which aim at reducing the depth, i.e., the number of levels, or the size of graph, i.e., the number of nodes, respectively. For area rewriting, [19] suggests a set of axioms called *eliminate* including $\Omega.M$; $\Omega.D_{R\to L}$. *eliminate* can remove some of the MIG nodes by repeatedly applying majority rule ($\Omega.M$) and distributivity from right to left ($\Omega.D_{R\to L}$) to the entire MIG. Assuming x, y, z, u, and v as input variables $\Omega.D_{R\to L}$ transforms $M(M(x, y, u), M(x, y, v), z)$ to $M(x, y, M(u, v, z))$ which means the total number of nodes has decreased from three to two.

In general, the depth of the graph is of high importance in MIG optimization to lower the latency of the resulting circuits. The depth of the MIG can be reduced by pushing the critical variable with the longest arrival time to upper levels. For this purpose, a set of axioms called *push-up* has been proposed in [19]. *Push-up* includes majority, distributivity, and associativity axioms applied in a sequence, i.e., $\Omega.M$; $\Omega.D_{L\to R}$; $\Omega.A$; and $\Psi.C$. It is obvious that the majority rule may reduce depth by removing unnecessary nodes. Applying distributivity from left to right ($\Omega.D_{L\to R}$) such that $M(x, y, M(u, v, z))$ is transformed to $M(M(x, y, u), M(x, y, v), z)$ may also result in an MIG with smaller depth at a cost of one extra node. If either x or y is the critical variable with the latest arrival, distributivity cannot reduce the depth of $M(x, y, M(u, v, z))$. However, if z is the critical variable, applying $\Omega.D_{L\to R}$ will reduce the depth of MIG by pushing z one level up. In the cases that the associativity rules ($\Omega.A$, $\Psi.C$) are applicable, the depth can be reduced by one if the axioms move the critical variable to the upper level. After performing *push-up*, the relevance axiom ($\Psi.R$) is applied to replace the reconvergent variables that might provide further possibility of depth reduction for another *push-up*.

Using RRAM devices for implementation, considerable parts of the metrics determining area and delay depend on the number and distribution of complemented edges which are not intended in conventional area and depth optimization. We propose a multiobjective MIG optimization algorithm to obtain efficient RRAM-based logic circuits with a good tradeoff between both objectives. Algorithm 1 includes a combination of conventional area and depth optimization algorithms besides techniques tackling complemented edges from both aspects of area and delay and iterates them for a maximum number of cycles called *effort*. The algorithm starts with applying *push-up* to obtain a smaller depth. Then, the complemented edges are aimed by applying an extension of axiom inverter propagation from right to left ($\Omega.I_{R\to L}$) for the condition that the considered node has at least two outgoing complemented edges. The three cases satisfying this condition and their equivalent

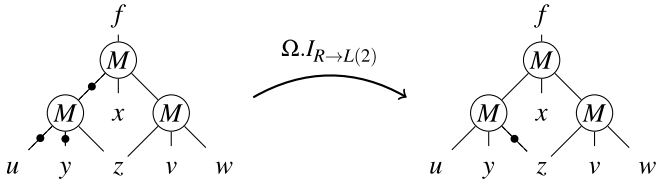


Fig. 6. Applying an extension of $\Omega.I_{R\rightarrow L}$ to reduce the extra RRAM devices and steps caused by complemented edges.

majority gates are shown below and discussed in the following considering their effect on both cost metrics:

$$M(\bar{x}, \bar{y}, \bar{z}) = \overline{M}(x, y, z) \quad (1)$$

$$\overline{M}(\bar{x}, \bar{y}, z) = M(x, y, \bar{z}) \quad (2)$$

$$M(\bar{x}, \bar{y}, z) = \overline{M}(x, y, \bar{z}). \quad (3)$$

In the first case, the ingoing complemented edges of the gate are decreased from three to zero, while one complement attribute is moved to the upper level, i.e., the level including the output of the gate. Assuming that the current level, i.e., the level including the ingoing edges, is the critical level with the maximum number of required RRAM devices, this case is favorable for area optimization. However, if the upper level is the critical level, the number of required RRAM devices will increase by only one. Similar scenarios exist for the two other cases, although the last case might be less interesting because the number of complemented edges in both levels is changed equally by one. That means a penalty of one is possible as the cost for a reduction of one, while transformations (1) and (2) may reduce the number of RRAM devices by three and two, respectively.

To reduce the number of operations, the number of levels possessing complemented edges should be reduced. Depending on the presence of complemented edges by other gates in both levels, the two first transformations given above might reduce or increase the number of operations or even leave it unchanged. Case (1) is beneficial if the upper level already has complemented edges and also the transformation removes all the complemented edges from the current level. It might be also neutral if none of the levels are going to be improved to a complement-free level. The worst case occurs when moving the complement attribute to the upper level increments the number of levels with complemented edges. Similar arguments can be made for the remaining cases. However, case (2) is more favorable because it never adds a level with complemented edges and case (3) cannot be advantageous because it can never release a level from complemented edges.

Fig. 6 shows a simple MIG that is applicable to transformation (2) ($\Omega.I_{R\rightarrow L(2)}$). The transformation has released one level of the MIG from the complement attribute (black dot), which results in a smaller number of computational steps. Furthermore, as a result of removing one complemented edge from the critical level, the required number of RRAM devices is decreased by one.

After applying inverter propagation for the aforementioned conditions ($\Omega.I_{R\rightarrow L(1-3)}$), the MIG is also reshaped and more chances for reducing the depth might be created. Thus, *push-up* is applied to the entire MIG again to reduce the number of operations as much as possible. In the last step, the number of RRAM devices are reduced to make a tradeoff between both objectives. Applying $\Omega.A$, some of changes by *push-up* that have increased the maximum level size can be undone.

Algorithm 1 MIG Rewriting for RRAM Cost Optimization

for (cycles = 0; cycles < effort; cycles++) **do**

$\Omega.M_{L\rightarrow R}$; $\Omega.D_{L\rightarrow R}$; $\Omega.A$; $\Psi.C$;
$\Omega.I_{R\rightarrow L(1-3)}$;
$\Omega.M_{L\rightarrow R}$; $\Omega.D_{L\rightarrow R}$; $\Omega.A$; $\Psi.C$;
$\Omega.A$; $\Omega.D_{R\rightarrow L}$;
} push-up

end for

Algorithm 2 MIG Rewriting for Operation Count Minimization

for (cycles = 0; cycles < effort; cycles++) **do**

$\Omega.M_{L\rightarrow R}$; $\Omega.D_{L\rightarrow R}$; $\Omega.A$; $\Psi.C$;
$\Omega.I_{R\rightarrow L}$;
$\Omega.I_{R\rightarrow L(1-3)}$;
$\Omega.M_{L\rightarrow R}$; $\Omega.D_{L\rightarrow R}$; $\Omega.A$; $\Psi.C$;
} push-up

end for

Finally, distributivity from right to left ($\Omega.D_{R\rightarrow L}$) is applied to the graph to reduce the number of nodes in levels.

Due to the importance of latency in logic synthesis, and the issue of sequential implementation in RRAM-based circuits, we propose Algorithm 2 for reducing the number of operations. In the proposed operation minimization algorithm, two axioms of inverter propagation are applied to the MIG after *push-up*. First, only the axiom presented by case (1), i.e., the base rule of inverter propagation from right to left ($\Omega.I_{R\rightarrow L}$), is applied to the entire MIG to lower the number of levels with complemented edges. Since the transformation moves one complement attribute to the upper level, it might create new inverter propagation candidates for the all three discussed cases if the upper level already has one or two ingoing complemented edges. Hence, we apply $\Omega.I_{R\rightarrow L(1-3)}$ again to ensure maximum coverage of complemented edges. Although case (3) cannot reduce the number of operations, it is not excluded from $\Omega.I_{R\rightarrow L(1-3)}$ due to its effect on balancing the levels' sizes. Finally, *push-up* is applied to the MIG to reduce the depth more if new opportunities are generated. It should be noted that the number of operations is mainly determined by the MIG depth. In fact, in the worst case caused by complemented edges, the total number of operations would be equal to seven times the number of levels, i.e., the MIG depth. Nevertheless, we show the efficiency of our proposed step optimization algorithm in the following section.

Algorithm 3 is proposed to reduce the number of required RRAM devices [35]. The algorithm starts with *eliminate* to reduce the number of nodes. Then, it applies $\Omega.A$, $\Psi.C$ to reshape the MIG to enable further reduction of area and applies *eliminate* for the second time as suggested in [19]. After eliminating the unnecessary nodes, we use $\Omega.I_{R\rightarrow L(1-3)}$ to reduce the number of additional RRAM devices required for complemented edges. At the end, since the MIG might have been changed after the three inverter propagation transformations, $\Omega.I_{R\rightarrow L}$ is applied again to ensure the most costly case with respect to the complemented edges is removed. In general, applying the last two lines of Algorithm 3 over the entire MIG repetitively can lead to much fewer RRAM cost.

4) Results of MIG-Based Synthesis: Table V shows the results of experiments performed to evaluate the three proposed MIG rewriting algorithms. The number of iterations

TABLE V
RESULTS OF MIG-BASED SYNTHESIS FOR RRAM-BASED IN-MEMORY COMPUTING USING THE THREE PROPOSED MIG OPTIMIZATION ALGORITHMS

	RRAM costs optimization				Operation minimization				RRAM device minimization			
	IMP		MAJ		IMP		MAJ		IMP		MAJ	
Benchmark	*R*	*OP*	*R*	*OP*	*R*	*OP*	*R*	*OP*	*R*	*OP*	*R*	*OP*
5xp1_90	199	99	149	36	264	77	182	28	141	175	97	63
alu4_98	2160	176	1370	72	2461	165	1717	56	1252	318	878	115
apex1	3676	165	2343	56	4335	121	2972	44	2410	329	1682	119
apex2	531	143	358	56	653	132	435	47	288	341	202	124
apex4	4728	143	2820	64	5340	132	3602	48	3454	286	2414	104
apex5	1482	141	1053	47	1975	98	1286	35	1176	284	816	102
apex6	1652	121	1018	44	1742	99	1191	36	1124	253	786	92
apex7	408	132	277	48	526	121	348	44	300	220	200	80
b9	252	87	168	32	252	66	168	28	252	121	168	44
clip_124	312	110	217	40	380	99	275	36	218	187	152	68
cm150a	147	77	95	32	132	88	90	32	132	143	88	52
cm162a	90	86	60	30	90	66	65	24	90	117	60	40
cm163a	102	76	68	27	102	66	68	24	102	97	68	34
cordic_138	189	121	134	48	229	99	162	39	144	248	96	87
misex1_178	111	66	76	24	130	55	94	20	92	99	64	36
misex3_180	2207	165	1444	67	2621	143	1762	52	1303	307	913	111
parity	216	132	152	53	216	154	152	48	111	275	77	100
seq_201	3189	153	1970	64	3551	132	2498	60	1747	341	1217	124
t481_208	148	142	90	52	188	110	123	40	102	241	68	87
table5	2630	154	1723	64	3393	142	2252	52	1579	363	1107	132
too_large	510	164	322	64	587	121	392	48	234	341	162	124
x1	569	99	435	36	711	77	509	28	322	176	226	64
x2	66	76	46	26	94	66	68	24	66	98	44	35
x3	1729	99	1008	44	1787	99	1201	36	1110	231	772	84
x4	599	77	391	28	694	66	563	24	570	121	380	44
AVG	1116.08	120.16	711.48	46.16	1298.12	103.76	887	38.12	732.76	228.48	509.48	82.6

Algorithm 3 MIG Rewriting for RRAM Device Minimization

for (cycles = 0; cycles < effort; cycles++) **do**

$\left.\begin{array}{l}\Omega.M;\ \Omega.D_{R\rightarrow L};\\ \Omega.A;\ \Omega.C;\\ \Omega.M;\ \Omega.D_{R\rightarrow L};\end{array}\right\}$ eliminate

$\Omega.I_{R\rightarrow L(1-3)};$

$\Omega.I_{R\rightarrow L};$

end for

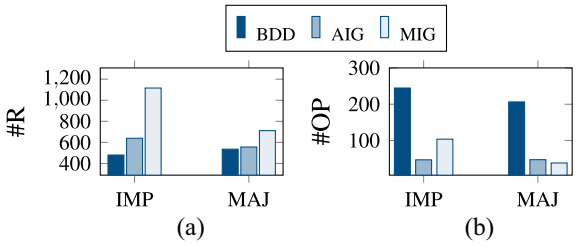


Fig. 7. Comparison of synthesis results by logic representations for RRAM-based in-memory computing. (a) Average number of RRAMs. (b) Average number of operations.

(effort) was set to 40. We observed that the MIGs are well optimized after 40 cycles and the cost metrics do not change noticeably with more iterations. The total runtime for the entire benchmark set under this setting has been about 3 s.

As expected the cost metrics are much lower for MIGs using the MAJ-based realization. The results confirm that the proposed operation count and RRAM device minimization algorithms have achieved the smallest value for the corresponding optimization objective which has worsened the other cost metric. The number of RRAM devices and operations given by the proposed multiobjective algorithm are between the minimum and maximum boundaries found by the operation count and RRAM device minimization algorithms. This confirms the capability of the proposed MIG rewriting technique to find a good tradeoff between both objectives.

More precise comparisons for the results of the MAJ-based realization show that the number of RRAM devices by the bi-objective algorithm is on average 19.78% less than that of the operation minimization algorithm at a cost of 21.09% increase in the number of operations. A similar comparison with results obtained by the RRAM device minimization algorithm shows an average reduction of 44.11% in the number of operations at a fair cost of 39.64% in the number of required RRAM devices.

IV. COMPARISON OF LOGIC REPRESENTATIONS

Fig. 7 compares the average values of synthesis results over the whole benchmark set for the three discussed logic representations. For a fair comparison and due to the high importance of latency in logic-in-memory computing synthesis, the values shown in Fig. 7 are chosen from optimization results with respect to the number of operations, i.e., MOGA with priority to the number of operations for BDDs, MIG rewriting for operation count minimization, and AIG depth rewriting.

According to Fig. 7, BDDs clearly achieve smaller number of RRAM devices and therefore can be a better choice when area is considered a more critical cost metric. On the other hand, the operation counts obtained by the BDD-based synthesis are much higher than the same values resulted by AIG-based and MIG-based methods. Comparison of synthesis results by the AIGs and MIGs also shows that the average number of operations for the MIG-based method using the MAJ-based realization is reduced by 19.37% compared to the AIG-based synthesis using depth minimization. This confirms the advantage of MIGs in providing higher speed in-memory computing circuits in comparison with the two other representations.

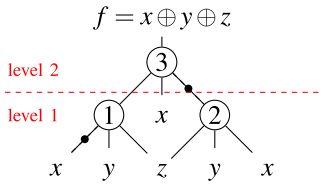


Fig. 8. MIG representing a three bit XOR gate.

V. Crossbar Implementation for the Proposed Synthesis Approach

We have already discussed the number of required RRAMs and operations for the presented in-memory computing approach using any of the three mentioned logic representations. In this section, we will show how an entire graph can be computed on a crossbar and what determines its required dimensions. We present step by step implementation of an example MIG shown in Fig. 8 which represent a three input XOR gate.

Both MAJ-based and IMP-based implementations for in-memory computing logic circuits can be executed on a standard crossbar architecture as shown in Fig. 9(a). In such an architecture, an entire row should be allocated for computing a single graph node which represents a data structure primitive, e.g., a majority gate if MIG is used for synthesis. To compute an entire level of a data structure simultaneously, all nodes of the level should be computed in parallel in separate rows. This means that a number of rows equal to the maximum level size in the entire graph is required. For example, for a logic representation whose largest level has four nodes, a crossbar architecture with at least four rows is needed, independent of the type of the utilized representation or the basic operation of MAJ or IMP. Nevertheless, the number of RRAM devices at each row, i.e, the number of columns, is determined by the RRAM-based realization of the exploited data structure primitive and therefore absolutely depends on both of the aforementioned conditions.

In the following, we assume that all the primary inputs are already written in the memory.

A. MAJ-Based Implementation

The values given for MAJ-based implementation in Table I indeed present the upper bounds for the crossbar realization. According to Table I, the MAJ-based synthesis of the MIG shown in Fig. 8 can be realized using a maximum of nine RRAM devices, since the critical level needs 2×4 for its nodes and 1 more RRAM for the ingoing complemented edge. Also, each level can require three operations (2×3), which results in a total of eight operations for the whole MIG considering the presence of complemented edges at both levels.

Here, we show that the resulting MAJ-based implementation can be executed much more efficiently with respect to both time and area.

The crossbar with the upper bound dimensions for the MAJ-based implementation of 3-bit XOR gate is shown in Fig. 9(b). The MIG has a maximum level size of two, and accordingly the required crossbar needs two rows. Each row consists of four RRAM devices to compute a node and one additional device to be used in case of having a complemented edge. We assume that a maximum of two ingoing edges for an MIG node can be complemented after rewriting, from which one

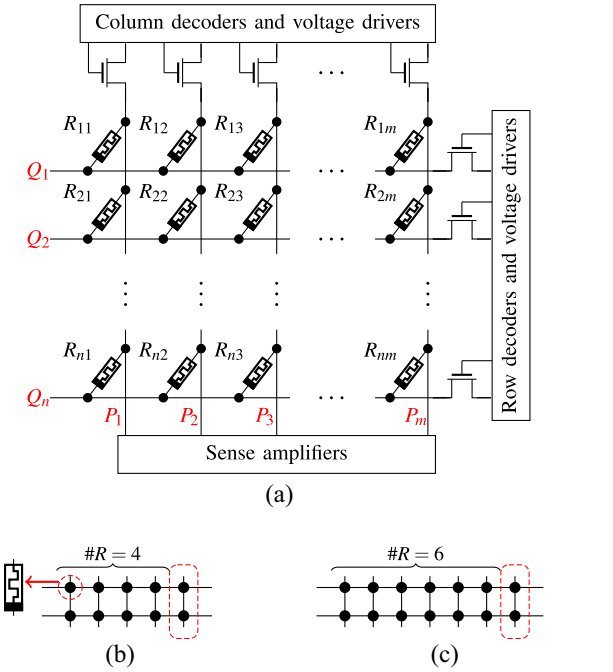


Fig. 9. Crossbar implementation for the presented synthesis approach for logic-in-memory computing. (a) Standard crossbar architecture. (b) Upper-bound crossbar for MAJ-based, and (c) IMP-based implementations for MIG shown in Fig. 8.

can directly be used as the second inverted operand of MAJ and thus, only one needs to be negated. The RRAM devices allocated for the complemented edges are displayed in red dashed surrounds at the end of the rows.

The implementation steps for the MAJ-based computation of the MIG shown in Fig. 8 are listed below

Initialization:	$R_{ij} = 0 : Q_{ij} = 1, P_{ij} = 0;$
1: Loading the third operands	$Q_1 = Q_2 = 0, P_1 = P_2 = z;$ $R_{11} : RM_3(z, 0, 0) = M(z, 1, 0) = z;$ $R_{21} : RM_3(z, 0, 0) = M(z, 1, 0) = z;$
2: Negation for node 2	$Q_1 = Q_2 = x, P_1 = x, P_2 = 1;$ $R_{25} : RM_3(1, x, 0) = M(1, \bar{x}, 0) = \bar{x};$
3: Computing level 1	node 1: $P_1 = y, Q_1 = x, R_{11} = z$ $R_{11} : RM_3(y, x, z) = M(y, \bar{x}, z);$ node 2: $P_1 = y, Q_2 = \bar{x}(@R_{25}), R_{21} = z;$ $R_{21} : RM_3(y, \bar{x}, z) = M(y, x, z);$
4: Computing level 2 (root node)	$P_1 = x, Q_1 = @R_{21}, R_{11} = M(\bar{x}, y, z);$ $R_{11} : RM_3(x, @R_{21}, @R_{11}) = M(x, \overline{@R_{21}}, @R_{11})$: $M(M(\bar{x}, y, z), x, \overline{M}(x, y, z)).$

We assume that all RRAM devices are first loaded with zero. For initialization, voltage levels 1 and 0 should be applied to the bottom electrodes (Q_{ij}) and the top electrodes (P_{ij}), respectively. This step is not considered in the operation count. Step 1 starts to compute the nodes 1 and 2 in level 1 (see Fig. 8) by loading the variable z as the third operands of MAJ. As said before, every node of the level should be computed in a separate crossbar row. Accordingly, nodes 1 and 2 are, respectively, computed in row 1 and 2 by selecting R_{11} and R_{21} as the corresponding third operands, i.e., the destinations of the operations. Then, the primary inputs are read from memory

and applied to the corresponding row and columns to execute the operations.

It is worth noting that the MAJ-based realization of majority gate in Section III-C1 also allocates RRAM devices for the first and the second operands. This is not actually required by the MAJ operation, dissimilar to IMP which needs all variables to be accessible on the same row. Nevertheless, all of the input RRAM devices are already considered in the crossbar with the upper bound dimensions shown in Fig. 9(b). Furthermore, in the MAJ-based realization of majority gate, we simply assumed that the second operand needs to be inverted and considered an RRAM device for it, while this is not always required. An MIG node with a single ingoing complemented edge can actually be implemented faster by skipping the negation and using the complemented edge directly as the second operand. In the MIG shown in Fig. 8, nodes 1 and 3 (the root node) are ideal for MAJ due to possessing a single complemented edge but node 2 requires one negation which needs to be performed first.

The negation required for node 2 is performed in step 2 at R_{25} by setting its bottom electrode (Q_2), to the value of x and its top electrode (P_2) to 1. It should be noted that R_{25} is not independently accessible and the entire second row and column are exposed to these voltage levels. Therefore, we need to ensure that the previously stored values are retained. By setting Q_2 to x, the bottom electrode of R_{21} is also changed to x. To maintain the value stored in R_{21}, its top electrode (P_1) should be also set to the same voltage level x as shown in step 2. By doing this, the top electrode of R_{11} changes to x, and thus its bottom electrode (Q_1) also needs to be set to x for keeping the current state of the devise.

The entire level 1 is computed simultaneously in step 3. One read is required to apply the value of $\bar{x}$ to Q_2. As shown in the step 3, both nodes can be computed in the same column since their first operands are equal, which is not necessarily true in all the cases. Step 4 computes the root node of the MIG. This can be done at one of the RRAM devices storing the intermediate results from the previous step and does not require any data loading. Since the value of node 2 is complemented, it is more efficient to use the value stored in R_{21} as the second operand to skip negation. This requires one read from R_{21} and R_{11} can be set to the third operand which is also the final destination of the computation.

B. IMP-Based Implementation

The required crossbar for the IMP-based implementation is shown in Fig. 9(c) which has one extra RRAM at the end of each row for the complemented edges. According to Table I, the example MIG shown in Fig. 8 with a maximum level size of 2 needs an upper bound of 12 (2×6) RRAM devices placed in two rows in addition to one more for the ingoing complemented edge. As Table I suggests, the computation needs 22 steps, 2×10 for the two levels plus two more steps for the complemented edges, including the IMP operations and the loads.

The required steps for the IMP-based implementation of the MIG shown in Fig. 8 are listed below

Initialization:	$R_{ij} = 0$;
1: Loading variables for level 1:	$R_{11} = x, R_{12} = y, R_{13} = z$; $R_{21} = x, R_{22} = y, R_{23} = z$;
2: Negation for node 1:	$R_{17} \leftarrow x$ IMP R_{17}: $R_{17} = \bar{x}$;
3-11: Computing level 1:	node1: $R_{14} = M(\bar{x}, y, z)$; node2: R_{24}: $M(x, y, z)$;
12: Loading variables for level 2:	$R_{11} = x, R_{12} = M(x, y, z), R_{13} = M(\bar{x}, y, z)$ $R_{14} = R_{15} = R_{16} = R_{17} = 0$;
13: Negation for node 3:	$R_{17} \leftarrow R_{12}$ IMP R_{17} : $R_{17} = \overline{R_{12}} = \overline{M}(x, y, z)$;
14-22: Computing level 2 (root node):	$R_{14} = M(M(\bar{x}, y, z), x, \overline{M}(x, y, z))$.

To explain the implementation step-by-step, we use names R_1 to R_6 for the RRAM devices at each row to denote X, Y, Z, A, B, and C, respectively, as used in Section III-C1. For initialization, all of the RRAM devices in the entire crossbar are cleared. Dissimilar to MAJ, IMP needs all variables used for computation to be stored in the same horizontal line. This means that there may be a need to have several copies of primary inputs or intermediate results at different rows simultaneously, as shown in step 1, where the variables of nodes 1 and 2 are loaded into RRAM devices in both rows. Step 2 computes the complemented edge of node 1 in the seventh RRAM device considered for this case at the end of first row, R_{17}. Steps 3–11 compute both nodes at level one and store the results in the forth RRAM device at the corresponding crossbar row, similarly to the RRAM device A used in Section III-C1.

The same procedure continues to compute the second level, which only consists of the MIG root node. Two out of the three inputs of node 3 are intermediate results, which have to be first read and then copied into the corresponding RRAM devices at row 1 besides other input and work devices as shown in step 12. In step 13, the complemented edge originating at node 2 is negated, and then root node is computed in step 22.

C. Discussion

The MAJ-based implementation for the example MIG in Fig. 8 was carried out using a small number of RRAM devices and within only four operations far less than the upper bounds, while no operation or RRAM devices could be saved during the IMP-based implementation. Length of operations required for data loading and negation of MAJ-based implementation can be even shortened much more for larger Boolean functions. Number of RRAM devices can also be much lower than the MAJ-based upper bounds given in Table I by performing operations successively in the devices carrying the results of previous levels.

It is obvious that using MAJ provides higher efficiency especially for MIG-based synthesis. Moreover, IMP requires additional voltages, which is not the case for MAJ, and as a result needs more complex control scheme and peripheral circuitry. However, MAJ-based implementation needs active read operations for each RM_3 cycle, while IMP-based implementation can reduce this requirement and propagate data natively within the memory array. MAJ-based implementation also does not allow to independently set the values of the top electrodes of the computing RRAM devices placed in the same columns. Such computation correlations do not occur during IMP-based operations since all IMP operations are executed with the same voltage levels V_{SET} and V_{COND}.

Nonetheless, dependency of the voltage levels of crossbar's rows and columns can be managed in many cases due to the commutativity property of the majority operation. This allows

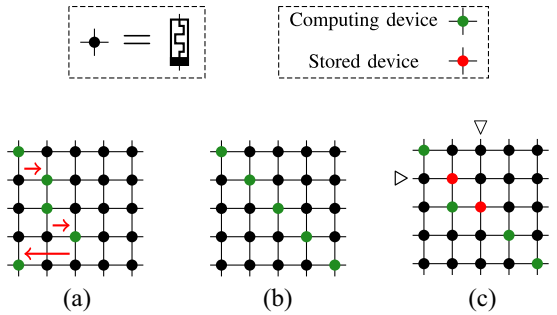


Fig. 10. Selecting crossbar computing RRAM devices to avoid computation interferences for MAJ-based implementation. (a) Performing conflicting computations at different columns. (b) Diagonal computation. (c) Retaining previously stored devices.

to perform computations simultaneously at RRAM devices in the same column if they share a single operand to be applied to the entire column. Using an automated procedure, the RRAM devices allocated for parallel computations can be placed in different nonconflicting columns as shown in Fig. 10(a). When none of the MAJ operations share any operand, the computations should be performed diagonally [Fig. 10(b)].

The rows or columns with computing devices may also possess previously stored RRAM devices, which values have to be maintained during computations by applying equal voltage levels to their top and bottom electrodes, i.e., their row and column drivers. For example, in Fig. 10(c), the second column has a stored device in the second row from top and a computing one in the third row. To keep the value of the stored device unchanged, the same voltage level, which is applied to its column for computing, has to be applied to its row. Setting the voltage level of the third column from left also needs a similar consideration as the column has a stored device located in a computing row.

It is obvious that a computation can coexist with the stored device in the same row or column if one of its operands is equal to what applied to the coordinates of the stored device. However, presence of several stored devices in a row or column may make it complex to arrange safe computations which can be handled by freeing such rows or columns from computation. As Fig. 10 shows, considerations regarding the conflicting computing or previously stored devices increase the area of the crossbar architecture since a larger number of columns or rows may be required, although the number of required RRAM devices does not change. Nevertheless, the number of steps can increase if the crossbar array does not meet the required number of rows and columns, which needs to move some computations into the successive steps.

VI. Conclusion

We presented an approach for logic synthesis of RRAM-based in-memory computing circuits using the logic representations BDDs, AIGs, and MIGs. We also showed that the presented approach provides valid and efficient crossbar implementations. The following remarks are concluded by comparison of experimental results.

1) The proposed BDD-based synthesis approach using multiobjective optimization reduces both cost metrics considerably compared to an existing BDD-based method. Using BDDs for synthesis results in smaller number of RRAM devices at a high cost in the number of operations in comparison with the two other logic representations.
2) The proposed AIG-based synthesis approach reduces the number of operations by an order of magnitude in comparison with an existing approach, as well as providing a fair tradeoff between both cost metrics among the experimented representations.
3) In comparison with BDDs and AIGs, MIGs show a high capability in reduction of the length of operations which is mostly considered as the main drawback of RRAM-based in-memory computing.
4) MAJ combined with the use of MIGs, provides a platform for logic-in-memory computing synthesis, which is highly efficient with respect to latency and crossbar dimensions.

References

[1] H.-S. P. Wong *et al.*, "Metal–oxide RRAM," *Proc. IEEE*, vol. 100, no. 6, pp. 1951–1970, Jun. 2012.
[2] L. Chua, "Memristor-the missing circuit element," *IEEE Trans. Circuit Theory*, vol. 18, no. 5, pp. 507–519, Sep. 1971.
[3] L. Chua, "Resistance switching memories are memristors," *Appl. Phys. A*, vol. 102, no. 4, pp. 765–783, 2011.
[4] S. H. Jo, K.-H. Kim, and W. Lu, "High-density crossbar arrays based on a Si memristive system," *Nano Lett.*, vol. 9, no. 2, pp. 870–874, 2009.
[5] Y. Ho, G. M. Huang, and P. Li, "Dynamical properties and design analysis for nonvolatile memristor memories," *IEEE Trans. Circuits Syst. I, Reg. Papers*, vol. 58, no. 4, pp. 724–736, Apr. 2011.
[6] K.-C. Liu *et al.*, "The resistive switching characteristics of a Ti/Gd_2O_3/Pt RRAM device," *Microelectron. Rel.*, vol. 50, no. 5, pp. 670–673, 2010.
[7] Y. V. Pershin and M. Di Ventra, "Practical approach to programmable analog circuits with memristors," *IEEE Trans. Circuits Syst. I, Reg. Papers*, vol. 57, no. 8, pp. 1857–1864, Aug. 2010.
[8] K.-T. T. Cheng and D. B. Strukov, "3D CMOS-memristor hybrid circuits: Devices, integration, architecture, and applications," in *Proc. ACM Int. Symp. Phys. Design*, Napa, CA, USA, 2012, pp. 33–40.
[9] P.-E. Gaillardon *et al.*, "A ultra-low-power FPGA based on monolithically integrated RRAMs," in *Proc. DATE*, Grenoble, France, 2015, pp. 1203–1208.
[10] D. B. Strukov, "Nanotechnology: Smart connections," *Nature*, vol. 476, pp. 403–405, Aug. 2011.
[11] J. Borghetti *et al.*, "'Memristive' switches enable 'stateful' logic operations via material implication," *Nature*, vol. 464, pp. 873–876, Apr. 2010.
[12] P.-E. Gaillardon *et al.*, "The programmable logic-in-memory (PLiM) computer," in *Proc. DATE*, Dresden, Germany, 2016, pp. 427–432.
[13] S. Kvatinsky *et al.*, "Memristor-based material implication (IMPLY) logic: Design principles and methodologies," *IEEE Trans. Very Large Scale Integr. (VLSI) Syst.*, vol. 22, no. 10, pp. 2054–2066, Oct. 2014.
[14] E. Lehtonen, J. Poikonen, and M. Laiho, "Implication logic synthesis methods for memristors," in *Proc. ISCAS*, Seoul, South Korea, 2012, pp. 2441–2444.
[15] N. Talati, S. Gupta, P. Mane, and S. Kvatinsky, "Logic design within memristive memories using memristor-aided loGIC (MAGIC)," *IEEE Trans. Nanotechnol.*, vol. 15, no. 4, pp. 635–650, Jul. 2016.
[16] H. Owlia, P. Keshavarzi, and A. Rezai, "A novel digital logic implementation approach on nanocrossbar arrays using memristor-based multiplexers," *Microelectron. J.*, vol. 45, no. 6, pp. 597–603, 2014.
[17] S. Chakraborti, P. V. Chowdhary, K. Datta, and I. Sengupta, "BDD based synthesis of Boolean functions using memristors," in *Proc. IDT*, Algiers, Algeria, 2014, pp. 136–141.
[18] J. Bürger, C. Teuscher, and M. Perkowski, "Digital logic synthesis for memristors," in *Proc. Reed Muller*, 2013.
[19] L. G. Amarù, P.-E. Gaillardon, and G. D. Micheli, "Majority-inverter graph: A novel data-structure and algorithms for efficient logic optimization," in *Proc. DAC*, San Francisco, CA, USA, 2014, pp. 1–6.
[20] L. Amarù *et al.*, "Majority-inverter graph for FPGA synthesis," in *Proc. SASIMI*, 2015, pp. 165–170.
[21] R. Drechsler and D. Sieling, "Binary decision diagrams in theory and practice," *Int. J. Softw. Tools Technol. Transf.*, vol. 3, no. 2, pp. 112–136, 2001.

[22] B. Bollig and I. Wegener, "Improving the variable ordering of OBDDs is NP-complete," *IEEE Trans. Comput.*, vol. 45, no. 9, pp. 993–1002, Sep. 1996.
[23] A. Kuehlmann, V. Paruthi, F. Krohm, and M. K. Ganai, "Robust Boolean reasoning for equivalence checking and functional property verification," *IEEE Trans. Comput.-Aided Design Integr. Circuits Syst.*, vol. 21, no. 12, pp. 1377–1394, Dec. 2002.
[24] L. G. Amarù, P.-E. Gaillardon, and G. De Micheli, "Majority-inverter graph: A new paradigm for logic optimization," *IEEE Trans. Comput.-Aided Design Integr. Circuits Syst.*, vol. 35, no. 5, pp. 806–819, May 2016.
[25] A. Chattopadhyay and Z. Rakosi, "Combinational logic synthesis for material implication," in *Proc. VLSI SoC*, Hong Kong, 2011, pp. 200–203.
[26] J. C. Beatty, "An axiomatic approach to code optimization for expressions," *J. ACM*, vol. 19, no. 4, pp. 613–640, 1972.
[27] S. Kvatinsky *et al.*, "MAGIC—Memristor-aided logic," *IEEE Trans. Circuits Syst. II, Exp. Briefs*, vol. 61, no. 11, pp. 895–899, Nov. 2014.
[28] Berkeley Logic Synthesis and Verification Group. *ABC: A System for Sequential Synthesis and Verification, Release 2016-02-09*. [Online]. Available: http://www.eecs.berkeley.edu/~alanmi/abc/
[29] S. Shirinzadeh, M. Soeken, and R. Drechsler, "Multi-objective BDD optimization for RRAM based circuit design," in *Proc. DDECS*, Košice, Slovakia, 2016, pp. 1–6.
[30] H. Li *et al.*, "A learnable parallel processing architecture towards unity of memory and computing," *Sci. Rep.*, vol. 5, Aug. 2015, Art. no. 13330.
[31] K. Deb, A. Pratap, S. Agarwal, and T. Meyarivan, "A fast and elitist multiobjective genetic algorithm: NSGA-II," *IEEE Trans. Evol. Comput.*, vol. 6, no. 2, pp. 182–197, Apr. 2002.
[32] S. Shirinzadeh, M. Soeken, and R. Drechsler, "Multi-objective BDD optimization with evolutionary algorithms," in *Proc. GECCO*, Madrid, Spain, 2015, pp. 751–758.
[33] S. Yang, *Logic Synthesis and Optimization Benchmarks User Guide: Version 3.0*, MCNC, Durham, NC, USA, 1991.
[34] S. Shirinzadeh, M. Soeken, P.-E. Gaillardon, and R. Drechsler, "Fast logic synthesis for RRAM-based in-memory computing using majority-inverter graphs," in *Proc. DATE*, Dresden, Germany, 2016, pp. 948–953.
[35] M. Soeken *et al.*, "An MIG-based compiler for programmable logic-in-memory architectures," in *Proc. DAC*, 2016, pp. 1–6.

Saeideh Shirinzadeh (S'16) received the B.Sc. and M.Sc. degrees in electrical engineering from the University of Guilan, Rasht, Iran, in 2010 and 2012, respectively. She is currently pursuing the Ph.D. degree with the Group of Computer Architecture, University of Bremen, Bremen, Germany.

Her current research interests include multiobjective optimization, evolutionary computation, logic synthesis, and in-memory computing.

Mathias Soeken (S'09–M'13) received the Diploma degree in engineering and the Ph.D. degree in computer science and engineering from the University of Bremen, Bremen, Germany, in 2008 and 2013, respectively.

He is a Scientist with the Integrated Systems Laboratory, École Polytechnique Fédéderale de Lausanne, Lausanne, Switzerland. He is involved in active collaborations with the University of California at Berkeley, Berkeley, CA, USA, and Microsoft Research, Redmond, WA, USA. He is also actively maintaining the logic synthesis frameworks CirKit and RevKit. His current research interests include the many aspects of logic synthesis and formal verification, constraint-based techniques in logic synthesis, and industrial-strength design automation for quantum computing.

Dr. Soeken was a recipient of the Scholarship from German Academic Scholarship Foundation, the Germany's oldest and largest organization that sponsors outstanding students in the Federal Republic of Germany. He has been serving as a TPC member for several conferences, including Design Automation Conference'17 and IEEE/ACM International Conference on Computer-Aided Design (ICCAD)'17. He is a Reviewer for Mathematical Reviews as well as for several journals.

Pierre-Emmanuel Gaillardon (S'10–M'11–SM'16) received the Electrical Engineering degree from CPE-Lyon, Villeurbanne, France, in 2008, the M.Sc. degree in electrical engineering from INSA Lyon, Villeurbanne, in 2008, and the Ph.D. degree in electrical engineering from CEA-LETI, Grenoble, France, and the University of Lyon, Lyon, France, in 2011.

He was a Research Assistant with CEA-LETI. He was a Research Associate with the Swiss Federal Institute of Technology, Lausanne, Switzerland, within the Laboratory of Integrated Systems (Prof. De Micheli), and a Visiting Research Associate with Stanford University, Palo Alto, CA, USA. He is an Assistant Professor with the Electrical and Computer Engineering Department, University of Utah, Salt Lake City, UT, USA, and he leads the Laboratory for NanoIntegrated Systems. His current research interests include development of reconfigurable logic architectures and digital circuits exploiting emerging device technologies, and novel EDA techniques.

Prof. Gaillardon was a recipient of the C-Innov 2011 Best Thesis Award and the Nanoarch 2012 Best Paper Award. He is an Associate Editor of the IEEE TRANSACTIONS ON NANOTECHNOLOGY. He has been serving as a TPC member for many conferences, including Design, Automation & Test in Europe (DATE)'15-16, Design Automation Conference'16, Nanoarch'12-16. He is a Reviewer for several journals and funding agencies. He will serve as the Topic Co-Chair "Emerging Technologies for Future Memories" for DATE'17.

Rolf Drechsler (M'94–SM'03–F'15) received the Diploma and Dr.Phil.Nat. degrees in computer science from Johann Wolfgang Goethe University Frankfurt am Main, Frankfurt, Germany, in 1992 and 1995, respectively.

He was with the Institute of Computer Science, Albert-Ludwigs University, Freiburg im Breisgau, Germany, from 1995 to 2000, and with the Corporate Technology Department, Siemens AG, Munich, Germany, from 2000 to 2001. Since 2001, he has been with the University of Bremen, Bremen, Germany, where he is currently a Full Professor and the Head of the Group for Computer Architecture, Institute of Computer Science. In 2011, he additionally became the Director of the Cyber-Physical Systems Group, German Research Center for Artificial Intelligence, Bremen. He is the Co-Founder of the Graduate School of Embedded Systems and he is the Coordinator of the Graduate School "System Design" funded within the German Excellence Initiative. His current research interests include the development and design of data structures and algorithms with a focus on circuit and system design.

Mr. Drechsler was a recipient of the best paper awards at the Haifa Verification Conference in 2006, the Forum on Specification & Design Languages in 2007 and 2010, the IEEE Symposium on Design and Diagnostics of Electronic Circuits and Systems in 2010, and the IEEE/ACM International Conference on Computer-Aided Design (ICCAD) in 2013. He is an Associate Editor of the IEEE TRANSACTIONS ON COMPUTER-AIDED DESIGN OF INTEGRATED CIRCUITS AND SYSTEMS, the *ACM Journal on Emerging Technologies in Computing Systems*, the *IET Cyber-Physical Systems: Theory & Applications*, and the *International Journal on Multiple-Valued Logic and Soft Computing*. He was a member of Program Committees of numerous conferences, including Design Automation Conference (DAC), ICCAD, Design, Automation & Test in Europe (DATE), Asia and South Pacific Design Automation Conference, FDL, ACM-IEEE International Conference on Formal Methods and Models for System Design, and Formal Methods in Computer-Aided Design, the Symposiums Chair of IEEE International Symposium on Multiple-Valued Logic 1999 and 2014, and the Topic Chair for "Formal Verification" DATE 2004, DATE 2005, DAC 2010, as well as DAC 2011. He is the General Chair of International Workshop on Logic Synthesis 2016 and the General Co-Chair of FDL 2016.