Logic Synthesis for In-Memory Computing using Resistive Memories

Saeideh Shirinzadeh*, Rolf Drechsler*†
*Department of Mathematics and Computer Science, University of Bremen, 28359 Bremen, Germany
†Cyber-Physical Systems, DFKI GmbH, 28359 Bremen, Germany

***Abstract*—The increasing urge to bypass the issue of the memory bottleneck in the current computer architectures has attracted high attention to in-memory computing enabled by emerging memory technologies such as *Resistive Random Access Memory* (RRAM). This paper studies in-memory computing from two perspectives, i.e. customized and instruction-based. The customized approach exploits logic representations to synthesize for in-memory computing. The approach proposes design methodologies and optimization algorithms for each representation with respect to area and latency upon the realizations of their logic primitives. The instruction-based approach proposes an automatic compiler to execute instructions on a logic-in-memory computer architecture and optimizes the programs. Experimental results for both approaches reveal considerable improvements compared to the state-of-the-art.**

Keywords*—*In-memory computing; RRAM; logic synthesis;

I. Introduction

The advancements in the processors of modern computers have far exceeded that of memory. The considerably higher latency of memory compared to processor on one hand, and the requirement of communication between these two in the von Neumann architecture on the other hand, has limited the overall performance of current computing systems which is known as *memory wall*. The growing need to deal with higher amount of data demanded by emerging applications such as *internet of things* (IoT) and *big data* has prompted much research to alleviate this problem [1]. Among these solutions, *in-memory computing* sounds to be very promising as it allows to go beyond the memory wall by integrating the storage and computing paradigms. This provides speedups of several orders of magnitude.

One of the core technologies enabling in-memory computing is *Resistive Random Access Memory* (RRAM). RRAM is a promising non-volatile memory technology with high scalability and zero standby leakage energy which internal resistance can be switched between two states, i.e. low and high. So far, several approaches have been proposed which exploit this resistive switching property to execute logic functions within RRAM devices exploiting. *Material Implication* (IMP) has been widely used for logic-in-memory computing [2]. A logic family called MAGIC was also proposed in [3] which allows to realize Boolean functions using NOR and NOT operations. In [4], it was shown that RRAM natively implements a majority oriented operation (MAJ) which allows to utilize majority logic for in-memory computing.

In this paper, we explore synthesis for in-memory computing from two different perspectives, i.e. (i) based on a customized approach at gate level which employs logic representations, and (ii) an instruction oriented approach for efficient manipulation, compilation, and execution of programs on a logic-in-memory architecture. The presented customized synthesis approach uses IMP and MAJ as basic operations on RRAM array which can be alternatively according to the design preferences, while the instruction based approach uses MAJ only for executing programs.

The customized synthesis approach starts with finding efficient realizations for the logic primitives of the employed representations, i.e. *Binary Decision Diagram* (BDD), *And-Inverter Graph* (AIG), and *Majority-Inverter Graph* (MIG). The approach presents optimization algorithms and a comprehensive design methodology to map each representation into equivalent netlists of operations and RRAM devices to be performed on a resistive memory array. The results of the presented customized approach reveal significant improvement compared to the state-of-the-art using the same logic representations.

By means of the instruction-based approach, we fully automatize and optimize an existing logic-in-memory computing architecture. Furthermore, we address the issue of lower write endurance of RRAM devices and propose wear-leveling techniques to increase the lifetime of the aforementioned architectures. Experiments performed on large arithmetic and control functions show considerable improvement in the distribution of writes all over the memory array as well as the number of cycles and RRAM devices representing the latency and area of the resulting implementations.

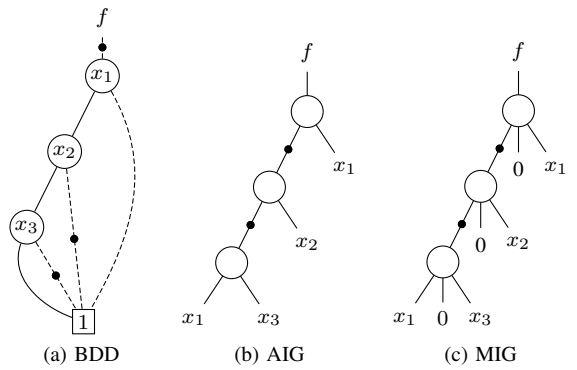
The remaining of this paper is structured as the following. To keep the paper self-contained, required preliminaries are explained in Section II in adition to a brief review of the state-of-the-art. Section III and Section IV present the customized and instruction-based synthesis approaches, respectively. Section V concludes the paper.

II. Background

A. Logic Representations

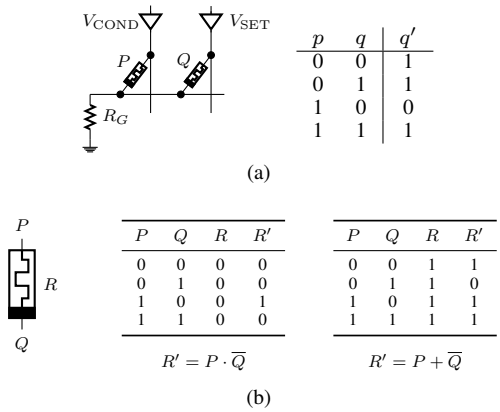
1) Binary Decision Diagrams (BDDs): *Binary Decision Diagram* (BDD, [5]) is a graph based data structure which allows to represent Boolean functions efficiently. A BDD is obtained from recursively applying Shannon decomposition $f = x_i f_{x_i} \oplus \bar{x}_i f_{\bar{x}_i}$ such that each node represents a sub-function. Using complemented edges in the graph, also allows to represent a sub-function and its complement by the same node. The decomposition is performed according to a certain variable ordering which results in a canonical BDD. For example, the ordering of the BDD shown in Fig. 1(a) is in order $x_1 < x_2 < x_3$, where the complemented edges are denoted by dots on the successors.

In circuit realization, each BDD node is mapped to a multiplexer which makes the number of nodes a determining factor in the costs of the resulting implementations. Therefore, BDD optimization, i.e. finding a variable ordering which results in a smaller BDD has been of high interest for applications utilizing BDDs. In this paper, we assume that the variable



(a) BDD (b) AIG (c) MIG

Fig. 1. Logic representations for an example function with three input variables.



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

(a)

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

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

(b)

Fig. 2. The Boolean operations executable within RRAM used at this work. (a) Implementation of IMP and its truth table [2]. (b) The intrinsic majority operation within an RRAM device [4].

ordering of an initial BDD before optimization is ascending $x_1 < x_2 < \cdots < x_n$, where n is the number of input variables.

2) Homogeneous Logic Representations for Circuits: *And-Inverter Graph* (AIG, [6]) and *Majority-Inverter Graph* (MIG, [7]) are homogeneous logic representation which are used in this work. An AIG node designates logical conjunction $x \cdot y$, which is majority of three $M(x, y, z) = x \cdot y + x \cdot z + y \cdot z$ in case of MIGs. The graphs also include inverters which are in terms of complemented edges. Both representations allow to efficiently represent Boolean functions and are utilized by state-of-the-art synthesis tools.

An AIG can easily be transformed into an MIG by adding a third zero input $x \cdot y = M(x, y, 0)$. Fig. 1(b) and (c) show both AIG and MIG representation for a three input function. As the figure shows, the number of nodes for both graphs is equal to three. However, MIGs can allow even more compact representations compared to AIGs [8].

B. Logic Operations enabled within RRAM

1) Material Implication (IMP): In [2], it was shown that material implication (IMP), i.e. $q' \leftarrow p \,\mathrm{IMP}\, q = \overline{p} + q$, can be executed from the interaction of two resistive switches under certain voltage levels shown by V_{SET} and V_{COND} in Fig. 2(a). The logical states of the resistive devices can also be simply switched between logic 1 or 0, i.e. FALSE operation, when applied to appropriate voltage pulses to set or clear the devices. IMP and FALSE together make a universal set of logic operations sufficient to execute arbitrary Boolean functions on resistive arrays.

2) Resistive Majority Operation (MAJ): In [4], an intrinsic majority operation enabled by RRAM devices was introduced which suffices to compute any Boolean function. Let us denote the top and bottom electrodes of an RRAM device with P and Q (see Fig. 2(b)). Assuming that the current resistive state of the device (R) can be switched to 1 and 0 by applying a positive or negative voltage level V_{PQ}, respectively, the next state of the device (R') changes based on the truth tables shown in Fig. 2(b). By expanding the Boolean relation in the tables, it can simply be shown that the next resistance state of the device is equal to $R' = M(P, \overline{Q}, R) = P \cdot \overline{Q} + P \cdot R + \overline{Q} \cdot R$, i.e. the result of three-input majority function when the logical state of the bottom electrode is inverted. This operation is referred to three-input resistive majority operation which is denoted by MAJ in this paper.

C. Related Work

The majority of related work for logic-in-memory synthesis using resistive devices exploit IMP as the basic operation. In [9], an approach was presented to map BDDs into networks of resistive devices. The work follows two main objectives in the presented mapping methodologies, one with respect to the area, i.e. serial computation, and the other with respect to the latency, i.e. parallel computation. In [10], a BDD-approach for logic-in-memory synthesis was proposed which improves the parallel methodology in [9]. [10] exploits a multi-objective BDD optimization algorithm which results in significantly improved results in terms of area and latency compared to [9]. AIG [11] and OIG [12], i.e. *Or-Inverter Graphs* have been also utilized for synthesis with RRAM devices using IMP operation.

Other basic operations enabled by RRAM devices have been also exploited in the state-of-the-art. MAGIC, i.e. *Memristor-Aided loGIC*, [3] is one of this approaches which allows to implement logic functions as a network of NOR gates on RRAM array. In [4], MAJ was used as the only form of logic operation performed in the *Programmable Logic-in-Memory* (PLiM) architecture. The architecture was simply designed to control a single MAJ instruction during each cycle. The number of executable MAJ-based instructions per cycle was increased to two in [13] enabled by heuristic palatalization algorithms. Technology mapping for the architecture proposed in [13] was further improved in terms of crossbar area [14] as well as delay [15] by applying optimization techniques during technology mapping.

III. Customized Synthesis

The customized synthesis approach includes three stages, (i) finding efficient realizations with RRAM devices for logic primitive of each representation, (ii) then, defining the design methodology to map the representations into RRAM array, (iii) and finally optimizing the representations with respect to the number of RRAM devices and operations determined by the design methodology. In the following, we explain the proposed customized approach for logic representations BDD and AIG briefly and provide an implementation example in the case of MIG. Due to the lack of space, we refer the reader to [16] for the details of the optimization algorithms for each representation.

An IMP-based realization for a 2-to-1 multiplexer (MUX) has been proposed in [9] which requires six operations and five RRAM devices. Using MAJ, the realization [16] requires six devices and executes the MUX function within five operations. This trade-off between the required number of RRAM devices

TABLE I. THE 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 devices	
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 : $K=3$

and operations allows to choose between the realizations according to the design preferences when either latency or area is of higher importance.

Table I shows the cost metrics of the BDD-based synthesis approach for both IMP and MAJ-based realizations. The BDD is first optimized with respect to these cost metrics, see optimization algorithms in [10], [16], and can then be implemented on an RRAM array according to a level-by-level design methodology. This means that starting from the bottom of the graph all of the nodes in each BDD level are computed simultaneously. After computation of each level, the RRAM devices are released and can be used as input devices for computation of the next level. This procedure continues until the root function is computed.

For computing each level, the number of required RRAM devices includes five or six times the number of nodes in the level, one extra RRAM device for each complemented edge, and one extra device for preserving the value of each nonconsecutive fanout, i.e. a fanout targeting a level else than the next successive one. Since the RRAM devices are reused, the number of devices required to compute a BDD is equal to the maximum number of required devices over all of the levels. The number of operations to compute a BDD is at least six or five rimes the number of BDD levels, for IMP or MAJ-based representations, respectively. However, this value must be added to the number of levels which possess complemented edges as their negation needs extra operations. It should be noted that copying the values of the nonconsecutive fanouts can be performed at the same time step of loading the inputs of the level, and therefore is not counted in the number of operations [16].

Table I also shows the cost metrics for AIG-based synthesis obtained by the same level-by-level design methodology explained above. The values are shown for the *imp*-based realization of NAND gate [11] and the MAJ-based realization of AND gate [16] as logic primitives of AIG. Both realizations need three RRAM devices and three operations. The IMP-based realization also includes an inverter, which can be directly used for the complemented edges but needs inversion

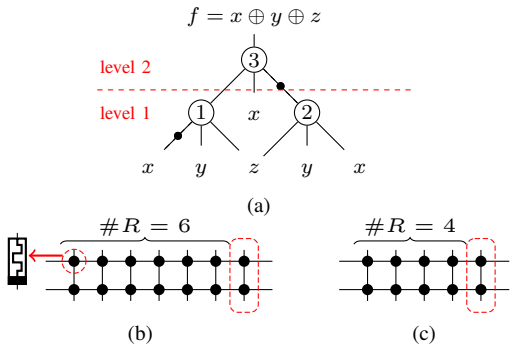


Fig. 3. (a) MIG representing a three bit XOR gate, and upper-bound crossbar for implementing it using (b) IMP-based and (c) MAJ-based realizations.

for the regular edges. As a result, the levels possessing regular edges need additional operations.

The IMP-based realization [17], [16] for the majority gate is shown in the following. The realization needs six RRAM devices, input devices X, Y, and Z and extra devices A, B, and C required for negating or storing the operations' outputs. The majority function is executed after ten operations. The first operation loads the required devices with the input variables and zero, and the rest of the steps include IMP and FALSE operations.

01: $X = x, Y = y, Z = z$
$A = 0, B = 0, C = 0$
02: $a \leftarrow x$ IMP $a = \bar{x}$
03: $b \leftarrow y$ IMP $b = \bar{y}$
04: $y \leftarrow a$ IMP $y = x + y$
05: $b \leftarrow x$ IMP $b = \bar{x} + \bar{y}$
06: $c \leftarrow y$ IMP $c = \overline{x + y}$
07: $c \leftarrow z$ IMP $c = \overline{x \cdot z + y \cdot z}$
08: $a = 0$
09: $a \leftarrow b$ IMP $a = x \cdot y$
10: $a \leftarrow c$ IMP $a = x \cdot y + y \cdot z + x \cdot z$.

It is obvious that the MAJ-based realization for MIG-based synthesis can be realized more efficiently due to exploiting the natively implemented majority function in RRAM devices. As shown in the following, the realization of majority gate in this case needs a maximum of four devices and three steps.

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: $P_Z = x, Q_Z = \bar{y}, R_Z = z \Rightarrow R'_Z = M(x, y, z)$.

Fig. 3(a) shows an MIG representing a three-input XOR gate. Here, we show how this MIG can be implemented on an RRAM array with devices shown by R_{ij}, where i and j denote the indices of the row and column, respectively.

The presented design methodology computes all nodes in a level simultaneously and for this purpose it allocates a row to each node. This means a single operation per cycle is performed at each row. As the example MIG has a maximum level size of two, the required crossbar needs at least two rows with at least six devices (see Fig. 3(b)). Also, one extra device at the end of each row is considered to be used for complemented edges. As Table I predicts the number of required steps is 22, i.e. 10 times the depth 2 plus two additional steps for the complemented edges at both levels.

Initialization	$R_{ij} = 0$;
1: Loading 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	node 1: $R_{14} = M(\bar{x}, y, z)$; node 2: $R_{24} : M(x, y, z)$;
12: Loading 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	$R_{14} = M(M(\bar{x}, y, z), x, \overline{M}(x, y, z))$;

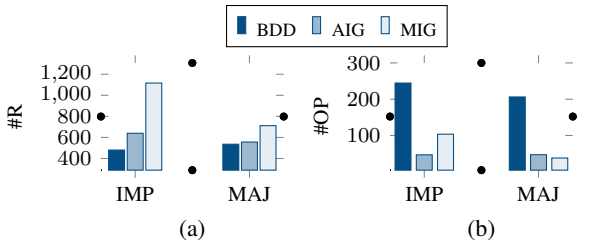


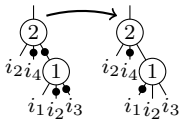
Fig. 4. Comparison of synthesis results by logic representations for RRAM-based in-memory computing, (a) the average number of RRAM devices, (b) the average number of operations.

The steps for the MAJ-based implementation of the MIG shown in Fig. 3(a) are shown in the following. As the steps show, the XOR function is executed using only three devices and within only four steps despite the upper bound of 8 which is predicted by Table I for an MIG with two levels possessing complemented edges. Indeed, the complemented edges at nodes 1 and 3 can be directly used as the second input of MAJ without being inverted. Furthermore, the RRAM updated devices can be used as inputs for the next cycle which means that the loading step is not required. It should be noted that applying signals to the rows and columns during the MAJ-based implementation should avoid data distortion by preserving previously computed results and the simultaneous operations in other rows [16]. For example, in step 2 the values of R_{11} and R_{21} are preserved by equalizing the logical states of their terminals when performing a MAJ operation within R_{25}.

Initialization	$R_{ij} = 0 : Q_{ij} = 1, P_{ij} = 0;$
1: Loading	$Q_1 = Q_2 = 0, P_1 = P_2 = z;$ $R_{11} : RM_3(z, 0, 0) = M(z, 1, 0) = z;$ $R_{21} : RM_3(z, 0, 0) = M(z, 1, 0) = z;$
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	$P_1 = x, Q_1 = @R_{21}, R_{11} = M(\bar{x}, y, z);$ $R_{11} : RM_3(x, @R_{21}, @R_{11}) = M(x, \overline{@R_{21}}, @R_{11}) :$ $M(M(\bar{x}, y, z), x, \overline{M}(x, y, z));$

Table II shows the results of the proposed approach for the MIG-based synthesis using both IMP and MAJ and compares them with the state-of-the-art BDD-based [9] and AIG-based approach [11]. According to Table II, the total number of operations by our proposed MIG-based synthesis approach using MAJ-based realization is almost one-ninth of the corresponding value by BDD-based synthesis [9] at a fair cost of 57.42% increase in the number of RRAM devices. Even when the IMP-based realization is used our proposed approach is three times faster that that presented in [9]. Furthermore, in comparison to [11], the results show speed-ups of 7.1 and 2.57 times using maj and imp for implementation, respectively.

Fig. 4 compares the synthesis results based on the three representations using both IMP and MAJ for implementation. As the figure shows, the BDD-based approach needs the smallest number of devices but leads to high latency. On the other hand, synthesis methodologies based on AIG and MIG require more RRAM devices but decrease the length of operations. In particular, the MIG-based approach using MAJ results in implementations with the smallest latency.



Before optimization		After optimization	
1: 0, 1, @R_1	$R_1 \leftarrow 0$	**1:** 0, 1, @R_1	$R_1 \leftarrow 0$
2: 1, i_3, @R_1	$R_1 \leftarrow \bar{i}_3$	**2:** i_3, 0, @R_1	$R_1 \leftarrow i_3$
3: i_1, i_2, @R_1	$R_1 \leftarrow N_1$	**3:** i_2, i_1, @R_1	$R_1 \leftarrow N_1$
4: 0, 1, @R_2	$R_2 \leftarrow 0$	**4:** i_4, i_2, @R_1	$R_1 \leftarrow N_2$
5: 1, @R_1, @R_2	$R_2 \leftarrow \overline{N}_1$		
6: i_2, i_4, @R_2	$R_2 \leftarrow N_2$		

Fig. 5. The effect of MIG optimization on the number of RRAM devices and instructions.

IV. Instruction-based Synthesis

In [4], a *Programmable Logic-in-Memory* (PLiM) computer architecture was proposed which allows to perform logic operations on a regular RRAM array when its controller is on. The PLiM can also perform as a standard RAM system in case that the control signal is off. The controller consists of a simple finite state machine and some work registers to perform MAJ operations, which we refer to instructions. Only a single MAJ instruction is allowed per cycle. An arbitrary instruction has the format $M(P, \bar{Q}, R)$ with three operands to be assigned. The first operand P is the signal applied to the top electrode of the RRAM device, i.e. the row driver, and the second operand Q is the signal applied to the bottom electrode, i.e. the column driver. The third operand R is the current state of the device under computation which updates automatically when the instruction is executed.

The PLiM computer derives the instruction set for computing an arbitrary Boolean function from its MIG representation. The characteristics of the MIG, the order of nodes for computation, and assigning the operands for each instruction significantly affect the number of required RRAM devices and instructions which address the area and delay of the resulting PLiM implementations [1], [18], [19]. In this work, we propose an automatic compiler which generates PLiM programs for computing arbitrary functions while addressing the aforementioned cost metrics.

As the PLiM performs computations fully serially the number of nodes in the MIG, i.e. the size of the MIG, is a determining factor in the length of the resulting set of instructions. Therefore, optimizing the MIG with respect to the number of nodes can considerably improve the latency of PLiM implementations. However, the number of nodes is not the only feature that should be addressed during MIG optimization. MAJ ideally needs one complemented edge to compute a node within a single instruction. Otherwise, one extra device and instruction are required to invert one operand before computing the node which makes the number and position of the complemented edges influential. Fig. 5 shows an example MIG with two nodes before and after optimization which has only changed the graph with respect to the complemented edges. As the figure shows, MIG optimization reduces both of the number of RRAM devices and instructions required for implementation. We refer the reader to [18] for the MIG optimization algorithm for PLiM.

As mentioned before, different orders of nodes for computation as well as choosing the operands result in different cost metrics regarding area and latency. Our proposed compiler first finds an efficient order of candidate nodes for computation, and then translates the nodes into a MAJ instruction by assigning the operands with the smallest number of RRAM devices and instructions.

The candidate selection procedure starts with listing the nodes which are computable, i.e. their children nodes are already computed. Then, it ranks the candidate nodes according

TABLE II. COMPARISON OF RESULTS BY THE PROPOSED CUSTOMIZED APPROACH WITH EXISTING APPROACHES USING BDD [9] AND AIG [11]

Benchmark	PI	BDD [9]		MIG-IMP		MIG-MAJ		Benchmark	Inputs	AIG [11]	MIG-IMP		MIG-MAJ	
		$\#R$	$\#OP$	$\#R$	$\#OP$	$\#R$	$\#OP$			$\#OP$	$\#R$	$\#OP$	$\#R$	$\#OP$
5xp1_90	7	84	73	199	99	149	36	9sym_d	9	1418	923	175	398	60
alu4_98	14	642	334	2160	176	1370	72	con1f1	7	18	70	75	28	26
apex1	45	1626	705	3676	165	2343	56	con2f2	7	19	60	76	24	24
apex2	39	122	237	531	143	358	56	exam1_d	3	12	43	44	19	16
apex4	9	2073	447	4728	143	2820	64	exam3_d	4	12	50	55	20	23
apex5	117	806	888	1482	141	1053	47	max46_d	9	427	408	131	193	48
apex6	135	770	1169	1652	121	1018	44	newill_d	8	50	129	109	57	40
apex7	49	290	437	408	132	277	48	newtag_d	8	21	90	96	36	33
b9	41	125	298	252	87	168	32	rd53f1	5	27	60	64	24	25
clip	9	120	89	312	110	217	40	rd53f2	5	57	77	77	35	28
cm150a	21	56	**127**	147	77	95	32	rd53f3	5	32	86	66	38	24
cm162a	14	46	102	90	86	60	30	rd73f1	7	238	291	121	140	44
cm163a	16	42	116	102	76	68	27	rd73f2	7	46	129	88	57	32
cordic	23	32	149	189	121	134	48	rd73f3	7	104	193	107	84	39
misex1	8	83	69	111	66	76	24	rd84f1	8	351	430	153	187	52
misex3	14	444	185	2207	165	1444	67	rd84f2	8	47	172	88	76	31
parity	16	23	113	216	132	152	53	rd84f3	8	23	90	50	36	15
seq	41	1566	692	3189	153	1970	64	rd84f4	8	345	473	141	214	47
t481	16	26	107	148	142	90	52	sao2f1	10	102	110	108	72	35
table5	17	580	168	2630	154	1723	64	sao2f2	10	112	234	119	98	42
too_large	38	282	232	510	164	322	64	sao2f3	10	380	325	143	143	55
x1	51	230	398	569	99	435	36	sao2f4	10	252	326	143	163	59
x2	10	60	80	66	76	46	26	sym10_d	10	1172	1475	187	643	72
x3	135	770	1169	1729	99	1008	44	t481_d	16	1564	1285	187	567	72
x4	94	401	642	599	77	391	28	xor5_d	5	32	86	66	38	24
$\sum$		**11299**	9026	27902	3004	17787	**1154**	$\sum$	194	6861	7615	2669	**3390**	**966**

to their effects on the number of RRAM devices and the length of the instructions and finally chooses the best node for computation. The comparison of the nodes is performed based on two main principles, (i) increasing the number of devices which are released after computation of the node, and (ii) lowering the duration of time that RRAM devices are blocked and cannot be reused.

Fig. 6(a) shows an MIG with two candidate nodes u and v. One child node is shared between both of the candidates, which is also needed as an input of the root node and therefore the RRAM device keeping its value cannot be released after computing u and v. Hence, u has two releasing children, while v has only one. In this case, the compiler selects u to be computed first.

For the second principle, see Fig. 6(b). The node A is an input of the root node G. Thus, the RRAM device storing the value of A is blocked for long time and cannot be released to end of the computations. To avoid an increase in the number of RRAM devices resulted by this, the compiler computes such nodes with long waiting time as late as possible. This case, also causes an uneven write traffic which in the long term wears some devices out much earlier than others. Indeed, postponing to compute the nodes with long waiting time is a wear leveling strategy which enhances the lifetime by balancing the writes all over the memory [20]. This especially matters as RRAM devices have limited write endurance that should be addressed in the design process.

Besides the endurance-aware compilation, we also propose other techniques to uniformly distribute the writes. This includes endurance-aware MIG optimization to lower the sources of unbalanced write traffic within an MIG, and direct techniques, i.e. allocating the RRAM device with the minimum write count when a device is requested, and allocating fresh RRAM devices when all of previously freed devices have been already rewritten for more than a certain value [20].

Table III shows the number of required RRAM devices and instructions of the proposed instruction-based approach on a EPFL benchmarks[1]. The results are for shown for the

[1]http://lsi.epfl.ch/benchmarks

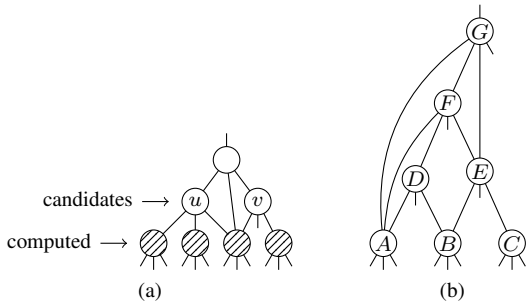


Fig. 6. (a) Reducing the number of RRAMs by selecting the candidate with more releasing children. (b) Reducing the number of RRAMs and balancing the write traffic by selecting the nodes with long storage duration later.

naïve PLiM implementations, implementations after only MIG optimization, and both MIG optimization and compilation. The results show improvements of 20.09% and 14.83%, respectively in the number of instructions and RRAM devices after only MIG optimization. As the compilation aims to reduce the number of the devices, the reduction in the total number of devices considerably improves to 61.4% after compilation.

The standard deviation of writes over the RRAM devices is shown in Fig. 7. The results are shown for the combination of the minimum write strategy, endurance-aware MIG optimization, and endurance-aware compilation. The comparison of the results with the naïve implementations shows an improvement of 72.17% over all of the benchmarks.

V. CONCLUSION

This paper presents a comprehensive customized approach for synthesis of logic-in-memory circuits using the logical representations BDD, AIG, and MIG. The presented approach introduces realization of the logic primitives using two basic operations enabled by RRAM devices, and provides optimization algorithms and design methodologies for crossbar implementation. The paper also proposes an automatic compiler for a regular logic-in-memory computer architecture and improves

TABLE III. EXPERIMENTAL EVALUATION OF THE PROPOSED INSTRUCTION-BASED APPROACH

Benchmark	PI/PO	naïve			MIG optimization					MIG optimization and compilation			
		$\#N$	$\#I$	$\#R$	$\#N$	$\#I$	impr.(%)	$\#R$	impr.(%)	$\#I$	impr.	$\#R$	impr.(%)
adder	256/129	1020	2844	512	1020	2037	28.38	386	24.61%	1911	32.81	259	49.41
bar	135/128	3336	8136	523	3240	5895	27.54	371	29.06%	6011	26.12	332	36.52
div	128/128	57247	146617	687	50841	147026	-0.03	771	-12.22%	147608	-0.68	590	14.12
log2	32/32	32060	78885	1597	31419	60402	23.43	1487	6.89%	60184	23.71	1256	21.35
max	512/130	2865	6731	1021	2845	5092	24.35	867	15.08%	4996	25.78	579	43.29
multiplier	128/128	27062	76156	2798	26951	56428	25.91	1672	40.24%	56009	26.45	419	85.03
sin	24/25	5416	12479	438	5344	10300	17.09	426	2.73%	10223	18.08	402	8.22
sqrt	128/64	24618	60691	375	22351	47454	21.81	433	-15.46%	49782	17.97	323	13.87
square	64/128	18484	54704	3272	18085	33625	38.53	3247	0.76%	33369	39.00	452	86.19
cavlc	10/11	693	1919	262	691	1146	40.28	236	9.92%	1124	41.43	102	61.07
ctrl	7/26	174	499	66	156	258	48.29	55	16.66%	263	47.29	39	40.91
dec	8/256	304	822	257	304	783	4.74	257	0.00%	777	5.47	258	-0.39
i2c	147/142	1342	3314	545	1311	2119	36.05	487	10.64%	2028	38.81	234	57.06
int2float	11/7	260	648	99	257	432	33.33	83	16.16%	428	33.95	41	58.59
mem_ctrl	1204/1231	46836	113244	8127	46519	85785	24.25	6708	17.46%	84963	24.97	2223	72.65
priority	128/8	978	2461	315	977	2126	13.61	241	23.49%	2147	12.76	149	52.70
router	60/30	257	503	117	257	407	19.09	112	4.27%	401	20.28	64	45.30
voter	1001/1	13758	38002	1749	12992	25009	34.19	1544	11.72%	24990	34.24	1063	39.22
$\sum$		236710	608655	22760	225560	486324	20.09	19383	14.83	487214	19.95	8785	61.40

$\#N$: number of MIG nodes, $\#I$: number of instructions, $\#R$: number of RRAM devices, improvement is calculated compared to naïve

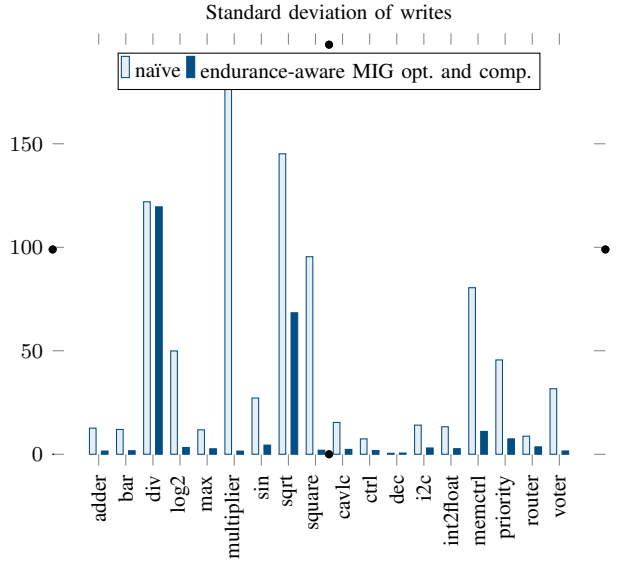


Fig. 7. Standard deviation of writes for the PLiM architecture.

the execution costs considerably with respect to latency, area, and the write balance.

ACKNOWLEDGMENTS

This research was supported by the University of Bremen's graduate school SyDe funded by the German Excellence Initiative.

REFERENCES

[1] M. Soeken, P.-E. Gaillardon, S. Shirinzadeh, R. Drechsler, and G. De Micheli, "A PLiM computer for the internet of things," *IEEE Computer*, vol. 50, no. 6, pp. 35–40, 2017.

[2] J. Borghetti, G. Snider, P. Kuekes, J. Yang, D. Stewart, and R. Williams, "Memristive switches enable stateful logic operations via material implication," *Nature*, vol. 464, no. 7290, pp. 873–876, 2010.

[3] S. Kvatinsky, D. Belousov, S. Liman, G. Satat, N. Wald, E. Friedman, A. Kolodny, and U. Weiser, "MAGIC – Memristor-Aided Logic," *IEEE Trans. Circuits Syst. II*, vol. 61, no. 11, pp. 895–899, 2014.

[4] P.-E. Gaillardon, L. G. Amarù, A. Siemon, E. Linn, R. Waser, A. Chattopadhyay, and G. De Micheli, "The programmable logic-in-memory (PLiM) computer," in *Design, Automation & Test in Europe*, 2016, pp. 427–432.

[5] R. Drechsler and D. Sieling, "Binary decision diagrams in theory and practice," *STTT*, vol. 3, no. 2, pp. 112–136, 2001.

[6] 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, 2002.

[7] L. G. Amarù, P.-E. Gaillardon, and G. De Micheli, "Majority-inverter graph: A novel data-structure and algorithms for efficient logic optimization," in *Design Automation Conference*, 2014, pp. 194:1–194:6.

[8] ——, "Majority-inverter graph: A new paradigm for logic optimization," *IEEE Trans. Comput.-Aided Design Integr. Circuits Syst.*, 2015.

[9] S. Chakraborti, P. Chowdhary, K. Datta, and I. Sengupta, "BDD based synthesis of Boolean functions using memristors," in *IDT*, 2014, pp. 136–141.

[10] S. Shirinzadeh, M. Soeken, and R. Drechsler, "Multi-objective BDD optimization for RRAM based circuit design," in *IEEE International Symposium on Design and Diagnostics of Electronic Circuits and Systems*, 2016, pp. 46–51.

[11] J. Bürger, C. Teuscher, and M. Perkowski, "Digital logic synthesis for memristors," in *Reed-Muller workshop*, 2013.

[12] A. Chattopadhyay and Z. Rakosi, "Combinational logic synthesis for material implication," in *IFIP/IEEE International Conference on Very Large Scale Integration (VLSI-SoC)*, 2011, pp. 200–203.

[13] D. Bhattacharjee, R. Devadoss, and A. Chattopadhyay, "ReVAMP: ReRAM based VLIW architecture for in-memory computing," in *Design, Automation and Test in Europe*, 2017, pp. 782–787.

[14] D. Bhattacharjee, A. Easwaran, and A. Chattopadhyay, "Area-constrained technology mapping for in-memory computing using ReRAM devices," in *Asia and South Pacific Design Automation Conference*, 2017, pp. 69–74.

[15] D. Bhattacharjee, L. Amarù, and A. Chattopadhyay, "Technology-aware logic synthesis for ReRAM based in-memory computing," in *Design, Automation & Test in Europ*, 2018, pp. 1435–1440.

[16] S. Shirinzadeh, M. Soeken, P.-E. Gaillardon, and R. Drechsler, "Logic synthesis for RRAM-based in-memory computing," *IEEE Trans. Comput.-Aided Design Integr. Circuits Syst.*, accepted for publication, 2018.

[17] ——, "Fast logic synthesis for RRAM-based in-memory computing using majority-inverter graphs," in *Design, Automation & Test in Europe*, 2016, pp. 948–953.

[18] M. Soeken, S. Shirinzadeh, P.-E. Gaillardon, L. G. Amarù, R. Drechsler, and G. De Micheli, "An MIG-based compiler for programmable logic-in-memory architectures," in *Design Automation Conference*, 2016, pp. 117:1–117:6.

[19] S. Shirinzadeh, M. Soeken, P.-E. Gaillardon, and R. Drechsler, "Logic synthesis for majority based in-memory computing," in *Advances in Memristors, Memristive Devices and Systems*, S. Vaidyanathan and C. Volos, Eds. Springer International Publishing, 2017, pp. 425–448.

[20] S. Shirinzadeh, M. Soeken, P.-E. Gaillardon, G. De Micheli, and R. Drechsler, "Endurance management for resistive logic-in-memory computing architectures," in *Design, Automation & Test in Europ*, 2017, pp. 1092–1097.