Multi-Objective BDD Optimization for RRAM based Circuit Design

Saeideh Shirinzadeh*, Mathias Soeken†, Rolf Drechsler*‡
*Department of Mathematics and Computer Science, University of Bremen, Germany
†Integrated Systems Laboratory, EPFL, Lausanne, Switzerland
‡Cyber-Physical Systems, DFKI GmbH, Bremen, Germany
{saeideh,drechsle}@cs.uni-bremen.de, mathias.soeken@epfl.ch

***Abstract*—Resistive switching property enables various promising applications such as design of non-volatile in-memory computing devices which has attracted high attention to *Resistive Random Access Memories* (RRAMs). In this work, we present a multi-objective BDD optimization approach for RRAM based logic circuit design. Dissimilar to classical BDD optimization, evaluating the cost metrics of the circuits in this case does not only depend on the number of BDD nodes but is more advanced. We have utilized a non-dominated sorting genetic algorithm for bi-objective BDD optimization with respect to the number of required RRAMs and computational steps addressing the area and delay of the resulting circuits, respectively. The algorithm also allows preference to one of the objectives if it is of higher significance. Experimental results show that the proposed multi-objective genetic algorithm achieves considerable reduction in both aforementioned criteria in comparison with an existing approach.**

I. Introduction

Binary Decision Diagrams (BDDs) are graph based data structures and are proven to be efficient for representing and manipulating Boolean functions. Electronic design automation and formal verification greatly benefit from BDDs in many applications. BDD optimization approaches especially node minimization techniques have been widely used in these applications. BDDs can be transferred directly to circuits by mapping each node to a multiplexer which makes the number of nodes the main cost metric. Nonetheless, there are few approaches which optimize BDDs with respect to other criteria such as the number of paths or expected and average path length [1]. Hence, in the majority of applications BDD optimization is classically defined as minimizing the number of BDD nodes which can be directly mapped to multiplexers to synthesize a circuit.

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 [2]. Nowadays, a variety of two-terminal devices based on resistance switching property exist which use different materials. In [3], *Resistive Random Access Memories* (RRAMs) were suggested as a physical implementation for the theory of memristors proposed in [4]. Although some researchers argued differences between memristor and RRAM, in this paper we use RRAM to refer to a generic resistive switching memory.

High scalibilty of RRAMs [2] makes it possible to implement ultra dense resistive memory arrays in hybrid nano/CMOS technology [5]. Such hybrid architectures using memristive devices are of high interest for their possible applications in non-volatile memory design, digital and analog programmable systems, and neuromorphic computing structures [6].

In [7], it was shown that *Material Implication* (IMP) can be executed by resistive switches to build logic circuits. In the same work a NAND gate was proposed that consists of three RRAMs and thus enables to realize any Boolean function. This allows advanced computer architectures different from classical von Neumannn architectures by providing memories able to perform computing operations [8]. So far, researchers have proposed various resistive logic circuits based on IMP operators. An RRAM based 2-to-1 multiplexer (MUX) containing six RRAMs was proposed in [6] that requires seven operations. In [9], a similar structure but more efficient in the number of RRAMs and operations was used for synthesis of Boolean functions using BDDs.

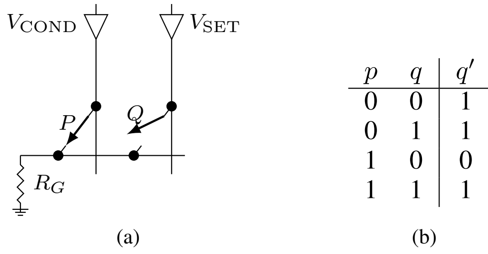
Although RRAM based implication logic is sufficient to express any Boolean function, the number of required computational steps to synthesize a given function is a real drawback [10]. So far, few works have been performed on the optimization of RRAM based in-memory computing circuits which aims at reducing the number of required steps and RRAMs. Besides BDDs, *And-Inverter Graphs* (AIGs) have been also used for logic synthesis with resistive memories [11]. However, none of these data structures have been optimized with respect to the cost metrics of in-memory computing circuit design. To the best of our knowledge, optimization for RRAM based design has been only performed on *Majority-Inverter Graphs* (MIGs) until now [12].

In this paper, BDDs using the MUX design proposed in [9] are optimized with respect to the number of RRAMs and computational steps which are related to the area and delay of the resulting circuits, respectively. The proposed multi-objective genetic algorithm minimizes both mentioned criteria for parallel evaluation where one MUX is considered for each node in a BDD level to synthesize the given function.

The remainder of this paper is organized as follows. Section II contains the basic principles of RRAM based logic and a brief description of the existing BDD optimization approaches. The proposed BDD optimization algorithm is discussed in Section III. Section IV presents the experimental results and Section V concludes the paper.

II. Background

This section describes the employed RRAM based 2-to-1 multiplexer circuit as well as discussing the state-of-the-art BDD optimization approaches. In order to keep the paper self-contained the implementation of the basic resistive IMP



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

Fig. 1. IMP operation. (a) Implementation of IMP using resistive switches. (b) Truth table for IMP ($q' \leftarrow p$ IMP q) [7]

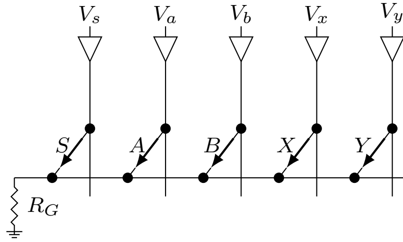


Fig. 2. RRAM based MUX circuit [9]

operation is also explained prior to introducing the MUX design.

A. RRAM based MUX

As mentioned before, the BDD nodes can be directly mapped to MUXes using resistive switches. The basic principles of resistive logic and the structure of an RRAM based MUX are explained in the following.

Any Boolean function can be expressed in one of the standard forms by using only *Material Implication* (IMP) and FALSE operation that always assigns the logic value 0. Fig. 1 demonstrates the basic implementation of an IMP gate including two resistive switches denoted by P and Q that are connected by a common horizontal nanowire to a load resistor R_G. Tri-state voltage drivers with a high-impedance output state for the undriven case control the vertical nanowires. Three voltage levels $\mathrm{V}_{\mathrm{SET}}$, $\mathrm{V}_{\mathrm{COND}}$ and $\mathrm{V}_{\mathrm{CLEAR}}$ are applied to perform IMP and FALSE operations by placing the switches in low-resistance state (logic 1) or high-resistance state (logic 0).

The FALSE operation can be executed by applying a positive voltage $\mathrm{V}_{\mathrm{CLEAR}}$ to a switch. A switch can be assigned to logic 1 by applying $\mathrm{V}_{\mathrm{SET}}$, i.e., a negative voltage larger than a threshold required to change the state of the driven device, to its voltage driver. $\mathrm{V}_{\mathrm{COND}}$ has a magnitude smaller than $\mathrm{V}_{\mathrm{SET}}$ which cannot cause any state change. Nonetheless, applying together $\mathrm{V}_{\mathrm{SET}}$ to Q and $\mathrm{V}_{\mathrm{COND}}$ to P simultaneously, the IMP operation can be implemented by interaction of two pulses through P, Q, and the load resistance. The conditional switching condition depends on the current logical states of p and q, such that switch Q is set to 1 if P is in high-resistance ($p = 0$) while it remains unchanged when P is in low-resistance ($p = 1$) [7].

Fig. 2 shows the RRAM based MUX proposed in [9]. The implementation requires six computational steps and five RRAMs of which three, named S, X and Y, are work RRAMs which initial values are replaced with intermediate results or the final output. The two other resistive switches, A and B, remain unchanged during operations and are called input RRAMs [9]. The corresponding implication steps of the MUX realization shown in Fig. 2 are as follows:

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

In the first step, a $\mathrm{V}_{\mathrm{CLEAR}}$ is applied to resistive switches X and Y to execute the FALSE operation. At the same time, the other three switches are also prepared by applying appropriate pulses, $\mathrm{V}_{\mathrm{SET}}$ or $\mathrm{V}_{\mathrm{CLEAR}}$, to their voltage drivers in order to initialize their desired initial states. The remaining steps are performed by sequential IMP operations that are executed by applying simultaneous voltage pulses $\mathrm{V}_{\mathrm{COND}}$ and $\mathrm{V}_{\mathrm{SET}}$.

B. BDD optimization methods

BDDs are a representation for Boolean functions that are canonical for a fixed variable ordering. BDD optimization is the task of finding a variable ordering which minimizes the considered cost metrics, e.g., the number of nodes in the BDD. Improving the variable ordering to find the optimum BDD is NP-complete [13].

BDD optimization is mainly defined as minimization of the size of diagram, that is the number of BDD nodes. Exact methods are the only category of classical size-driven BDD optimization approaches that guarantee to determine the optimal variable ordering. Nonetheless, the high order of run-time and complexity is a serious drawback of these methods [1]. Sifting [14] is a well-known dynamic reordering based BDD minimization technique that aims at finding the best position of each input variable, assuming that the relative order of other variables does not change. For this purpose, the adjacent variables are swapped and the size of the resulting BDDs are recorded. Finally, the optimum BDD is characterized by the the variables fixed at positions which led to the BDD with the minimum number of nodes. Heuristic approaches such as *Simulated Annealing* (SA) [15] and *Genetic Algorithms* (GAs) [16] have shown better performance than sifting. SA starts with a randomly generated variable ordering. In each iteration, the current ordering is kept or replaced by a neighboring ordering based on a transition probability which allows uphill moves in order to escape a local minimum. In [16], GA is combined with sifting such that the initial population is first optimized by sifting. GA terminates if the optimal variable ordering remains unchanged after a certain number of generations. In the last step, the optimal ordering is sifted to further improve the resulting BDD's size if possible.

The number of paths has been also considered as an objective for BDD optimization due to its importance in some applications. *Modified Sifting* (MS) [17] is a path minimization method that is structurally similar to sifting with a different objective. An evolutionary algorithm has been also proposed in [18] for one-path optimization that has experimentally shown to be able to achieve higher degree of minimization in comparison with MS. In [19], a multi-objective evolutionary algorithm for BDD optimization was proposed to minimize the number of nodes and paths simultaneously. The algorithm has shown a good trade-off between both objectives without any loss of quality compared to the existing node or path minimization techniques.

III. BDD OPTIMIZATION FOR RRAM BASED DESIGN

Optimization of RRAM based BDDs in this work is carried out as a bi-objective problem aiming at minimizing the number of RRAMs and computational steps simultaneously, i.e., finding a trade-off between area and delay of the resulting circuits. For this purpose, we have exploited a non-dominated sorting based genetic algorithm that has been experimentally proven useful in multi-objective BDD optimization [19]. In this section, we first present how the objective functions, i.e., the number of required RRAMs and steps, are calculated. Then, we explain the framework of our algorithm and the employed variation operators.

A. *Objective functions*

In [9], BDDs were evaluated for two types of serial and parallel implementations for RRAM based design. Serial implementation requires only one RRAM based MUX which can be reused later by other BDD nodes. That means one MUX in addition to a number of RRAMs required for fanouts and complemented edges [20] suffice to build the BDD. Nonetheless, in the presence of complemented edges, the number of required steps to evaluate the Boolean function is greater than six times the number of BDD nodes. It was experimentally shown in [9] that the number of computational steps can rapidly increase for functions with a large number of inputs. In order to escape heavy delay penalties, BDDs are evaluated in parallel in this work.

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. Therefore, regardless of the possible fanouts and complemented edges in the BDD, the number of required RRAMs is five times the maximum number of nodes in any BDD level. Although, the number of RRAMs is increased in comparison with the serial evaluation, the number of computational steps can be remarkably lowered to the number of BDD levels, i.e., the number of input variables of the given function.

Table I shows how the objective functions are evaluated for optimization. However, the larger part of the objective functions are explained above, some additional RRAMs addressing complemented edges and fanouts are still required. Every complemented edge in the BDD requires a NOT gate to invert its logic value. As shown in step 2 discussed in the previous section, inverting a variable can be executed after an IMP operation with a zero loaded RRAM. Accordingly, for each MUX with a complemented input an extra RRAM 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 operation should be executed to complete the logic NOT operation. It is obvious that the required IMP 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 the number of BDD levels with ingoing complemented edges besides the basic value of level counts.

It is obvious that the RRAMs keeping the outputs of each BDD level can be assigned to the inputs of the next

TABLE I. OBJECTIVE FUNCTIONS

Symbol	Definition	Value
N	No. of input variables	Given
FO	Maximum no. of nonconsecutive fanouts in any BDD level	Given
CE_i	No. of ingoing complemented edges in the ith BDD level	Given
NL_i	No. of nodes in the ith BDD level	Given
L	No. of BDD levels with ingoing complemented edges	Given
R	No. of RRAMs	$\max_{0 \le i \le N}(5 \cdot NL_i + CE_i) + FO$
S	No. of computational steps	$6 \cdot N + L$

successive level and be reused without any loss of information. Nonetheless, the results of nodes targeting levels which are not right after their origin level might be lost during computations if their corresponding RRAMs are rewritten by the next operations. Thus, we consider extra RRAMs 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 RRAMs 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 RRAMs 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.

B. *Genetic algorithm framework*

Here, we explain the framework of our multi-objective BDD optimization algorithm. Genetic and more generally evolutionary algorithms are known as powerful search tools and are of high interest for multi-objective optimization especially for solving NP-complete problems such as BDD optimization. The employed BDD optimization algorithm is based on NSGA-II (*Non-dominated Sorting Genetic Algorithm*) [21] which has shown excellent performance for the optimization problems with two or three objectives.

In general, an evolutionary algorithm consists of iteratively applying two processes of selection and variation to a population of possible solutions, i.e., a subset of the search space. Selection decides which individuals in the population are good enough to be used as parents by variation operators for generating an offspring, as well as determining the survival strategy for filling the population of the next generation. In our genetic algorithm, we have utilized non-dominated sorting relation to discriminate solutions during selection. Based on *Pareto-dominance*, an individual x is said to dominate y if none of its objective functions are greater than the corresponding objective function in y, and x at least has one objective function smaller than the corresponding one in y. According to non-dominated sorting, every individual should be assigned to a fitness value representing its overall level of dominance in such a way that the first level of dominance contains non-dominated solutions which are not dominated by any other solution in the population.

The general framework of our *Multi-Objective Genetic Algorithm* (MOGA) is described in Algorithm 1. First, a population of size N consisting of variable orderings representing a set of BDDs is generated randomly. Then, the initial population is evaluated to assign the corresponding number of RRAMs and steps to each BDD. In step 3, the population is classified into non-dominating fronts shown by $\{F_1, F_2, \dots\}$ such that members of each front are incomparable based on dominance. Steps 6–18 are iterated for a certain number of generations.

The chosen parents for reproduction are results of a binary tournament selection, that selects the individual with lower fitness between two randomly chosen individuals. In step 6, an offspring Q_t with the same size as the current population P_t is created after applying evolutionary operators such as recombination and mutation to the selected parents. Thereafter, Q_t is evaluated and the union of the children and the current populations, R_t is sorted. Starting from the first non-dominating front, individuals are copied into the population of the next generation P_{t+1}. This procedure is continued until the size of the front is greater than the remaining slots in P_{t+1}. In this case, the front's members are sorted based on their density information and then P_{t+1} is filled with the first ordered lesser crowded individuals.

Algorithm 1 MOGA for BDD optimization

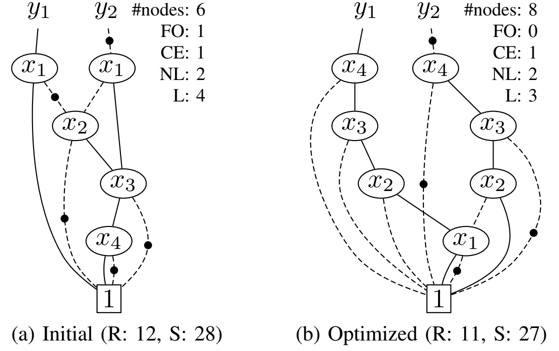
1: $P_0 \leftarrow$ InitializePopulation
2: EvaluateObjectiveFunctions(P_0)
3: $\{F_1, F_2, \ldots\} \leftarrow$ Non-dominatedSort(P_0)
4: $t \leftarrow 0$
5: **while** the stopping criterion is not met **do**
6: $Q_t \leftarrow$ MakeOffspringPopulation(P_{t+1})
7: EvaluateObjectiveFunctions(Q_t)
8: $R_t \leftarrow P_t \cup Q_t$
9: $\{F_1, F_2, \ldots\} \leftarrow$ Non-dominatedSort(R_t)
10: $P_{t+1} \leftarrow \emptyset$
11: $i \leftarrow 1$
12: **while** $|P_{t+1}| + |F_i| \leq N$ **do**
13: $P_{t+1} \leftarrow P_{t+1} \cup F_i$
14: $i \leftarrow i + 1$
15: **end while**
16: DistanceSort(F_i)
17: $P_{t+1} \leftarrow P_{t+1} \cup F_i[1 : (N - |P_{t+1}|)]$
18: $t \leftarrow t + 1$
19: **end while**

It might be desired to design smaller circuits at a fair cost of delay or vice versa. In [19], relation *priority dominance* was used in multi-objective BDD optimization to allow preference to the more significant objectives. Our genetic algorithm is enhanced with a priority vector which can be set by the user to perform optimization with preference to the number of RRAMs or steps.

C. Operators

In this section, the employed variation operators for creating the offspring population are briefly explained. We have utilized two recombination operators including *Partially Matched Crossover* (PMX) [22] and inversion which maintain validity of the variable permutations. PMX breaks selected parents into three sections by choosing two random positions. Then, two children are created by combining the sections from both parents in such a way that no previously used index is repeated inside the variable ordering. Applying inversion, a single parent is divided into three sections similarly to PMX. Finally, the order of variable indices in each section is inverted to produce a child.

The created children are also slightly changed by use of three mutation operators which are applied based on given probabilities. One mutation operator exchanges the contents of two randomly selected variable indices. The second mutation scheme performs the previous operator on the same child for two times. In the third employed mutation operator a random position in the child is selected and its value is exchanged for an adjacent index.



(a) Initial (R: 12, S: 28) (b) Optimized (R: 11, S: 27)

Fig. 3. Optimization example for MOGA

D. Example

Fig. 3 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 RRAMs for computing BDD levels $(5 \cdot NL + CE)$ is equal before and after optimization since both BDDs have a maximum number of two nodes and one ingoing complemented edge over all levels. However, there is a nonconsecutive fanout of node x_3 targeting x_1 before optimization requiring an extra RRAM 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 RRAMs by 1. The number of computational steps has been also reduced after optimization since one level has been released from complemented edges.

As can be seen, the numbers of RRAMs and steps 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.

IV. Experimental results

The results of experiments on 25 benchmark functions taken from LGSynth91 [24] are presented in this section. The number of input variables of the selected functions are in range from 7 to 135. We have used the CUDD package [23] for BDD representation and assessment of the optimization results. For each benchmark function, MOGA has been run 10 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 probabilities for PMX and inversion are set to 0.98 and 0.01 respectively. The mutation probability is distributed identically over the three operators with a value of $1/n$, where n is the number of input variables.

Table II compares the results of MOGA with the BDDs generated by the initial variable orderings given by CUDD.

TABLE II. COMPARISON OF OPTIMIZATION RESULTS WITH INITIALLY ORDERED BDDS BY CUDD [23]

Function	N	MOGA						CUDD [23]					
		FO	CE	NL	L	R	S	FO	CE	NL	L	R	S
5xp1	7	12	5	8	7	**57**	**49**	22	7	21	7	134	**49**
alu4	14	182	20	99	9	**697**	**93**	208	0	282	10	1618	94
apex1	45	278	41	164	12	**1139**	**282**	2885	477	1705	14	11887	284
apex2	39	45	0	24	1	**165**	**235**	2490	0	1910	13	12040	247
apex4	9	210	138	336	9	**2028**	**63**	40	175	405	9	2240	**63**
apex5	117	147	0	78	70	**537**	**772**	707	363	486	79	3500	781
apex6	135	49	0	26	4	**179**	**814**	281	151	284	75	1852	885
apex7	49	30	3	17	34	**118**	**328**	776	64	440	37	3040	331
b9	41	14	5	9	21	**64**	**267**	29	5	21	29	139	275
clip	9	18	5	14	9	**93**	**63**	16	11	69	9	372	**63**
cm150a	21	8	0	4	1	**28**	**127**	7650	0	65280	1	334050	**127**
cm162a	14	8	4	6	7	**42**	**91**	14	8	12	11	82	95
cm163a	16	6	3	4	16	**29**	112	9	8	12	14	77	110
cordic	23	5	0	4	2	**25**	**140**	5	0	4	19	25	157
misex1	8	19	0	12	2	**79**	**50**	4	5	11	8	64	56
misex3	14	101	0	67	7	**436**	91	117	0	192	2	1077	**86**
parity	16	0	1	1	16	**6**	**112**	0	1	1	16	**6**	**112**
seq	41	234	0	193	4	**1199**	**250**	10635	4801	18311	9	106991	255
t481	16	2	0	2	4	**12**	**100**	2	2	2	9	14	105
table5	17	162	0	101	7	**667**	109	435	0	450	8	2685	**110**
too_large	38	54	0	32	1	**214**	**229**	800	349	708	11	4689	239
x1	51	45	0	28	27	**185**	**333**	119	2	91	41	576	347
x2	10	7	3	7	5	**45**	**65**	4	8	19	6	107	66
x3	135	61	0	32	3	**221**	**813**	563	159	355	64	2497	874
x4	94	47	0	26	9	**177**	**573**	190	67	109	54	802	618
$\sum$		1744	228	1294	287	**8442**	**6161**	28001	6663	91180	555	490564	6429

N: no. of input variables, FO: maximum no. of nonconsecutive fanouts in any BDD level, CE: no. of ingoing complemented edges in the level determining the number of RRAMs, NL: no. of nodes in the level determining the number of RRAMs, L: no. of BDD levels with ingoing complemented edges, R: no. of RRAMs, S: no. of computational steps

Values given in Table II are the best found trade-off between objective functions chosen manually from the final populations of all 10 runs. The results in Table II show that BDDs found by MOGA require smaller number of RRAMs in comparison with the initially ordered BDDs. More precisely, the sum of RRAM counts by CUDD for the whole benchmark set is more than 58 times the corresponding amount of the optimized BDDs. The sum of the number of required steps to evaluate BDDs is also lowered by 4.16%. It should be noted that in parallel implementation the number of steps is close to the lowest possible value and does not depend on the BDD characteristics too much. As shown in Table I, the only cost metric that can be reduced by optimization is the number of levels with complemented edges. Therefore, the number of steps are not highly affected by optimization. A similar situation occurs in BDD optimization for serial implementation where the number of required RRAMs can be quite constant for a given function. Nonetheless, the optimization results show a considerable reduction in the values of objective functions.

MOGA is able to handle objective priorities when a fair increase in the area or delay can be tolerated for achieving higher minimization with respect to one criterion. Results of optimization with priority to the number of RRAMs and computational steps are demonstrated in Table III. As shown in the Table III, MOGA with priority to the number of RRAMs has obtained the smallest sum of RRAMs and as expected the smallest number of steps are provided by setting higher priority to steps. In both cases, a decrease in the objective function of higher importance has led to an increase in the value of the other objective. As discussed before, the number of required steps for evaluation cannot be dramatically lowered by optimization. This explains why MOGA with priority to the number of steps has resulted in higher overhead for a small reduction in S while a greater reduction in R has not increased the number of steps considerably.

In Table IV, our optimization results are compared with results given in [9]. The number of computational steps in results by Chakraborti et al. [9] are calculated by using a different function from ours given in Table I. In [9], the maximum number of complemented edges in any BDD level is added to the basic value relating to the number of input variables of the Boolean function. As explained before, we have considered one extra RRAM for any complemented edge in the diagram. Thus, for any level containing complemented edges one extra step is sufficient to execute all the required IMP operations simultaneously.

TABLE III. OPTIMIZATION RESULTS WITH PRIORITY TO THE NUMBER OF RRAMS OR STEPS

Function	Priority to R		Priority to S	
	R	S	R	S
5xp1	57	49	65	42
alu4	590	96	1014	77
apex1	1082	283	3040	277
apex2	172	237	188	235
apex4	2028	63	2224	62
apex5	509	775	591	771
apex6	160	815	220	813
apex7	102	332	190	328
b9	59	269	77	267
clip	93	63	93	63
cm150a	28	127	28	127
cm162a	32	93	38	89
cm163a	29	112	31	108
cordic	21	140	26	140
misex1	59	52	79	50
misex3	436	91	681	86
parity	6	112	6	112
seq	1129	251	1207	248
t481	12	100	16	100
table5	637	113	1346	107
too_large	164	232	182	229
x1	172	339	186	333
x2	42	67	45	65
x3	161	815	215	813
x4	177	573	209	573
$\sum$	**7957**	6199	11997	**6115**

R: no. of RRAMs, S: no. of computational steps

TABLE IV. COMPARISON OF RESULTS WITH CHAKRABORTI ET AL. [9]

Function	MOGA		Chakraborti et al. [9]	
	R	S	R	S
5xp1	**57**	**49**	84	73
alu4	697	**93**	**642**	334
apex1	**1139**	**282**	1626	705
apex2	165	**235**	**122**	237
apex4	**2028**	**63**	2073	447
apex5	**537**	**772**	806	888
apex6	**179**	**814**	770	1169
apex7	**118**	**328**	290	437
b9	**64**	**267**	125	298
clip	**93**	**63**	120	89
cm150a	**28**	**127**	56	**127**
cm162a	**42**	**91**	46	102
cm163a	**29**	112	42	116
cordic	**25**	**140**	32	149
misex1	**79**	**50**	83	69
misex3	**436**	**91**	444	185
parity	**6**	**112**	23	113
seq	**1199**	**250**	1566	692
t481	**12**	**100**	26	107
table5	**667**	**109**	580	168
too_large	**214**	**229**	282	232
x1	**185**	**333**	230	398
x2	**45**	**65**	60	80
x3	**221**	**813**	770	1169
x4	**177**	**573**	401	642
$\sum$	**8442**	**6161**	11299	9026

R: no. of RRAMs, S: no. of computational steps

A comparison of Table IV and Table II reveals that the results given in [9] do not represent the initially ordered BDDs. Actually, a kind of random optimization is done which tries a number of randomly created variable orderings and keeps the best found BDDs for each benchmark function. MOGA has achieved better performance in both objective functions. Regardless the difference in the number of steps which is slightly affected by the different equations for S, MOGA has also decreased the total number of required RRAMs. The sums of the number of RRAMs and steps required to evaluate BDDs of all benchmark function show a reduction of 25.28% and 31.74%, respectively.

V. CONCLUSION

RRAM based design has gained high interest for its possible applications in various domains. In logic circuit design using resistive switches, it is aimed to reduce the required number of RRAMs and computational steps. Especially the latter one can be more costly for larger circuits while using higher number of RRAMs does not necessarily cause high area overhead due to their tiny dimensions. The presented approach employs BDDs for RRAM based logic circuit design and performs multi-objective optimization by a genetic algorithm in order to attain smaller and faster circuits. The proposed genetic algorithm is capable of considering user preference to any of the objectives that might be of interest in some applications. Performance evaluation and comparison of experimental results reveal that our genetic algorithm fairly lowers both criteria and finds a good trade-off between them.

ACKNOWLEDGMENT

This research was supported by the German Research Foundation (DFG) (DR 287/23-1) within a Reinhart Koselleck project, the University of Bremen's graduate school SyDe, funded by the German Excellence Initiative, and H2020-ERC-2014-ADG 669354 CyberCare.

REFERENCES

[1] R. Ebendt, G. Fey, and R. Drechsler, *Advanced BDD Optimization*. Springer, 2005.

[2] H.-S. P. Wong, H. Lee, S. Yu, Y. Chen, Y. Wu, P. Chen, B. Lee, F. T. Chen, and M. Tsai, "Metal-oxide RRAM," *Proc. IEEE*, vol. 100, no. 6, pp. 1951–1970, 2012.

[3] L. Chua, "Resistance switching memories are memristors," *Applied Physics A*, vol. 102, no. 4, pp. 765–783, 2011.

[4] ——, "Memristor-The missing circuit element," *IEEE Trans. Circuit Theory*, vol. 18, no. 5, pp. 507–519, 1971.

[5] D. B. Strukov, D. R. Stewart, J. Borghetti, X. Li, M. Pickett, G. M. Ribeiro, W. Robinett, G. Snider, J. P. Strachan, W. Wu, Q. Xia, J. J. Yang, and R. S. Williams, "Hybrid CMOS/memristor circuits," in *ISCAS*, 2010, pp. 1967–1970.

[6] H. Owlia, P. Keshavarzi, and A. Rezai, "A novel digital logic implementation approach on nanocrossbar arrays using memristor-based multiplexers," *Microelectronics Journal*, vol. 45, no. 6, pp. 597–603, 2014.

[7] J. Borghetti, G. S. Snider, P. J. Kuekes, J. J. Yang, D. R. Stewart, and R. S. Williams, "Memristive switches enable stateful logic operations via material implication," *Nature*, vol. 464, pp. 873–876, 2010.

[8] S. Kvatinsky, G. Satat, N. Wald, E. Friedman, A. Kolodny, and U. Weiser, "Memristor-based material implication (IMPLY) logic: Design principles and methodologies," *IEEE Trans. VLSI Syst.*, vol. 22, no. 10, pp. 2054–2066, 2014.

[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] E. Lehtonen, J. Poikonen, and M. Laiho, "Implication logic synthesis methods for memristors," in *ISCAS*, 2012, pp. 2441–2444.

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

[12] S. Shirinzadeh, M. Soeken, P.-E. Gaillardon, and R. Drechsler, "Fast logic synthesis for RRAM-based in-memory computing using majority-inverter graphs," in *DATE*, 2016, pp. 948–953.

[13] B. Bollig and I. Wegener, "Improving the variable ordering of OBDDs is NP-complete," *IEEE Trans. Comput.*, vol. 45, no. 9, pp. 993–1002, 1996.

[14] R. Rudell, "Dynamic variable ordering for ordered binary decision diagrams," in *ICCAD*, 1993, pp. 42–47.

[15] B. Bollig, M. Löbbing, and I. Wegener, "Simulated annealing to improve variable orderings for OBDDs," in *International Workshop on Logic Synth*, 1995.

[16] R. Drechsler, B. Becker, and N. Göckel, "Genetic algorithm for variable ordering of OBDDs," *Proc. IEE Computers and Digital Techniques*, vol. 143, no. 6, pp. 364–368, 1996.

[17] G. Fey and R. Drechsler, "Minimizing the number of paths in BDDs: Theory and algorithm," *IEEE Trans. Comput.-Aided Design Integr. Circuits Syst.*, vol. 25, no. 1, pp. 4–11, 2006.

[18] M. Hilgemeier, N. Drechsler, and R. Drechsler, "Minimizing the number of one-paths in BDDs by an evolutionary algorithm," in *CEC*, 2003, pp. 1724–1731.

[19] S. Shirinzadeh, M. Soeken, and R. Drechsler, "Multi-objective BDD optimization with evolutionary algorithms," in *GECCO*, 2015, pp. 751–758.

[20] K. S. Brace, R. L. Rudell, and R. E. Bryant, "Efficient implementation of a BDD package," in *DAC*, 1990, pp. 40–45.

[21] 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, 2002.

[22] I. Oliver, D. Smith, and J. Holland, "Study of permutation crossover operators on the traveling salesman problem," in *Proc. the second International Conference on Genetic Algorithms*, 1987, pp. 224–230.

[23] F. Somenzi, "CUDD: CU Decision Diagram package release 2.5.0." University of Colorado at Boulder, 2012.

[24] S. Yang, "Logic synthesis and optimization benchmarks user guide: Version 3.0." Microelectronics Center of North Carolina (MCNC), 1991.