Logic Synthesis for Hybrid CMOS-ReRAM Sequential Circuits

Saman Froehlich Saeideh Shirinzadeh Rolf Drechsler

Cyber-Physical Systems, DFKI GmbH and Group of Computer Architecture, University of Bremen, Germany
{froehlich,s.shirinzadeh,drechsle}@cs.uni-bremen.de

***Abstract*—Resistive Random Access Memory (ReRAM) is an emerging non-volatile technology with high scalability and zero standby power which allows to perform logic primitives. ReRAM crossbar arrays combined with a CMOS substrate provide a wide range of benefits in logic synthesis.**

In this paper, we propose to exploit ReRAM in sequential circuits as it provides both required features as a computational and memory element. We propose a fully automated synthesis approach based on graph representations (i.e., BDDs and AIGs) for synthesis of sequential circuits on hybrid CMOS-ReRAM architectures. We propose an algorithm to efficiently divide the target function into two independent computational parts. This allows to merge part of the computation within a ReRAM unit and utilize its computational capabilities besides its function as a sequential element in order to minimize the CMOS overhead. Experimental results show that ReRAM allows for a significant reduction in CMOS size of up to 40.9% for BDDs with an average of 8.7% for BDDs and up to 10.1% with an average of 3.2% for AIGs.

I. Introduction

Resistive devices consisting of an oxide insulator layer placed between two metal electrodes have existed for several decades [1]. However, their promising properties have recently brought them into focus for their applications as memory or programmable computing elements. These devices predominantly known as *Resistive RAM* with acronyms RRAM or ReRAM allow for an abrupt switching of the electrical resistance between two high and low values which is maintained until changed by an appropriate voltage bias.

ReRAM as a memory technology is non-volatile, has a zero standby power, and is highly scalable. These features besides compatibility with conventional CMOS makes it possible to fabricate arrays of resistive memory devices on a CMOS substrate [1], [2]. Combining classical CMOS circuits with ReRAM layers in a hybrid structure has been studied before [2], [3]. However, hybrid architectures have not yet been utilized for the implementation of sequential logic circuits where ReRAM could be particularly beneficial because of possessing both computing and storing abilities. In this paper, we present the idea of exploiting a ReRAM network as a computational storage element instead of the conventional flip-flop used in current sequential circuits. Such a hybrid architecture allows smaller CMOS circuitry with less area and power dissipation. It is also possible to speed up the resulting implementations if the latency caused by computational steps of the ReRAM unit is less than the sum of the CMOS circuitry propagation delay and the sequential element clock cycle.

This work was supported in part by the German Research Foundation (DFG) within the project MANIAC (DR 287/29-1), by the German Federal Ministry of Education and Research (BMBF) within the project SELFIE under grant no. 01IW16001 and by the University of Bremen's graduate school SyDe, funded by the German Excellence Initiative.

An important distinction of this work with the related work is the automation of the synthesis process. To the best of our knowledge, all of the existing hybrid approaches use ReRAM devices within CMOS circuitries. In this work, we propose a comprehensive synthesis approach which fully automatizes the design process for hybrid CMOS-ReRAM logic circuits. Our approach provides separable CMOS and ReRAM units as shown in Fig. 1 which allows to use the state-of-the-art automated synthesis methodologies for both parts. To compute an arbitrary Boolean function, the proposed approach efficiently divides the computation between the CMOS circuitry (*CMOS part*) and the ReRAM network (*ReRAM part*) which also performs as the sequential element.

We propose a *node extraction* algorithm for graph-based synthesis, which aims at reducing the size of the CMOS circuitry. The proposed algorithm uses a modified subgraph matching algorithm to identify subfunctions. We extract these subfunctions to be computed within the ReRAM part such that the CMOS part of the circuit shrinks in size. The proposed node extraction algorithm is also integrated into existing tailored graph optimization algorithms. Our synthesis approach employs *Binary Decision Diagrams* (BDDs) and *And-Inverter Graphs* (AIGs) for efficient representation of target Booelan functions. Then, it applies the presented node extraction algorithm to the graphs to divide them into the CMOS and ReRAM computable parts and integrates graph-based optimization algorithms to optimize each part. It is worth noting that the presented approach is also applicable to combinational circuits. Nevertheless, the focus of this paper is on the sequential circuits in order to benefit from the memory aspect of ReRAM.

The main contributions of this paper are as follows:

- We present the general problem formulation for automated hybrid CMOS-ReRAM synthesis for the first time.
- In this context we present a novel node extraction algorithm, which allows to reduce the CMOS size by utilizing the ReRAM part for computation. For this algorithm, we also present tailored modifications for a general subgraph matching algorithm.
- The proposed approach integrates optimization techniques for BDDs and AIGs with respect to area and latency.

II. Related Work

There are several works which exploit different graph-based representations due to the higher efficiency for synthesis

using ReRAM. In [4], a BDD-based approach was proposed for synthesis with memristive devices. The approach presented in [4] includes two sequential and parallel evaluation techniques to compute BDD nodes realized by memristive circuits. BDDs were also utilized in [5], [6] with an improved design methodology and a multi-criteria optimization scheme to lower both latency and area of the resulting implementations simultaneously.

Other representations such as AIG [6], [7] and *Or-Inverter Graph* (OIG) [8] have been used by the state-of-the-art using material implication or the resistive majority operation. In [9], it was shown that the latter resistive operation is advantageous with respect to implementation costs in particular with respect to latency when used in a design procedure based on *Majority-Inverter Graphs* (MIGs). MIGs have also been employed for the representation and manipulation of instructions in logic-in-memory computer architectures where the tasks are processed and their results are stored within the same ReRAM array [10], [11].

In [6], a comprehensive synthesis procedure was presented which employs BDDs, AIGs and MIGs. This approach computes a graph in a level-by-level methodology, i.e. calculating all of the graph nodes at each level simultaneously starting from the bottom of graph. The calculations of each level is performed in several computational steps equal to the steps required for the execution of a single node realized with ReRAM devices. After computation of each level, the ReRAM devices are updated with the final node values. These values are directly used as the inputs of the next level. This procedure is applied recursively until finally the root node of the graph is computed. As the ReRAM devices can be reused by the consecutive levels, the total number of the devices required for computation depends on the most costly level which is determined by the number of nodes in level to a large extent. The latency, however, is mainly affected by the depth of the graph or the number of levels.

In this paper, we use the same approach proposed in [6] to implement the partial ReRAM circuit realizations of the BDDs and AIGs representing target functions. The basic operation enabled by the ReRAM is fixed to material implication.

III. Preliminaries

A. Logic Synthesis with ReRAM

In [12], it was shown that *material implication* $q = p \rightarrow q = \bar{p} + q$ can be executed by the electrical interaction of two ReRAM switches (memristors) under appropriate voltage bias. The implication operation together with a *false* operation, i.e. assigning the output to logical zero, form a universal set of logic primitives which suffice to execute any Boolean function.

Most of the related work on logic synthesis with resistive memories has been performed using material implication. However, other basic operations enabled by ReRAM devices have also been proposed which can be executed within one device, such as the *resistive majority operation* [10] or several devices, such as *Memristor-Aided Logic* (MAGIC) [13]. In this paper material implication [12] has been used as the memristive operation.

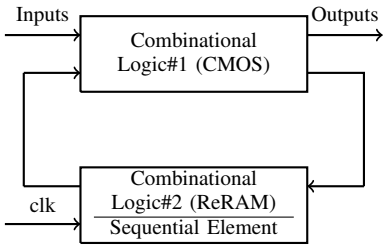


Fig. 1. Exploiting a ReRAM unit as partial combinational logic as well as sequential element in the proposed synthesis approach.

B. Graph-Based Representations

For the synthesis of Boolean functions, graph-Based representations are widely employed (e.g. [14], [15], [16]).

1) Binary Decision Diagrams (BDDs): *Binary Decision Diagrams* (BDDs) are graph-based representations of Boolean functions and are canonical for a given variable ordering if they are reduced and ordered. BDDs are based on the Shannon decomposition:

$$f = x_i f_{x_i} + \overline{x_i} f_{\overline{x_i}}$$

Since Brayton introduced efficient algorithms for BDD construction and manipulation in [17], they have become a state-of-the-art representation for Boolean functions in logic synthesis and verification. A BDD consists of nodes, which represent multiplexers and edges, which are the *true* and the *false* child of one such multiplexer. The input to each multiplexer is an input variable of the function.

In [6] a method to calculate the number of *ReRAM devices* (R_{BDD}) and the *number of operations* (OP_{BDD}) for a BDD-based implementation of a graph is given. The number of ReRAM devices and operations for the computation of a BDD-based implementation of a function using material implication as underlying function can be calculated as follows:

$$R_{BDD} = \max_{0 \leq i < D} (5 \cdot N_i + CE_i) + FO \qquad (1)$$
$$OP_{BDD} = 6 \cdot D + L_{CE}$$

here *D* is the number of levels in the BDD, N_i is the number of nodes, CE_i the number of incoming complimented edges to the corresponding level, *FO* the number of fan outs and L_{CE} is the number of levels with incoming complemented edges.

2) And Inverter Graphs (AIGs): An *And Inverter Graph* (AIG) is a representation of a Boolean network, where the edges represent wires between gates and the nodes represent logic gates (two input ANDs) or primary in- and outputs. Edges can be complimented to represent inverters. Eventhough AIGs are not canonical, their scalability makes them an efficient and popular choice.

To be able to implement AIGs in ReRAM, the authors of [6] have proposed a method to calculate the number of

ReRAM devices (R_{AIG}) and operations (OP_{AIG}) for an AIG-based implementation of a graph in ReRAM. R_{AIG} and OP_{AIG} using material implication as underlying function can be calculated as follows:

$$R_{AIG} = \max_{0 \leq i < D} (3 \cdot N_i + RE_i) \quad (2)$$
$$OP_{AIG} = 3 \cdot D + L_{RE}$$

here D is the number of levels in the AIG, N_i is the number of nodes, RE_i the number of incoming regular edges to the corresponding level and L_{RE} is the number of levels with incoming regular edges.

IV. Proposed Logic Synthesis Approach

In this section we present our proposed approach for Logic Synthesis of Hybrid CMOS-ReRAM Sequential Circuits. The main goal of our approach is to reduce the size of the CMOS part as much as possible, given a set of functions for which an implementation in the ReRAM part is known.

We give a problem formulation in Section IV-A. Successively, in Section IV-B, we present our proposed algorithm for the extraction of parts of the function which is to be synthesized from the CMOS part to the ReRAM part (*node extraction*). Finally, in Section IV-C and Section IV-D, we show how to apply our algorithm to BDDs and AIGs.

A. Problem Formulation

Given a graph-based representation $G_f(V_f, E_f)$ of a Boolean function $f(X)$ with $X = (x_1, \ldots, x_n) \in \mathbb{B}^n$ with a set of vertices V_f, corresponding vertex properties $vp(V_f)$ and edges E_f, corresponding edge properties $ep(E_f)$, a set of m Boolean functions $g = \{g_1, \ldots, g_m\}$ and an optimization algorithm opt, which is used to optimize G_f. During the application of opt extract all nodes, which can be represented by a concatenation c of $g_i \in g$ and a corresponding graph $G_c(V_c, E_c)$ with vertex properties $vp(V_c)$ and edge properties $ep(E_c)$, such that for each output of G_c there exists a matching output of G_f and for each edge $e_c \in E_c$ there exists a corresponding edge $e_f \in E_f$. The terminal vertices of G_c become the new output vertices of G_f.

For the given application, the Boolean function f is the function which is to be implemented in CMOS, if no computation is performed in the ReRAM part. The set g is the set of functions for which an implementation in ReRAM is known. Thus, we call the set g *ReRAM functions*. The vertex properties vp indicate if a node is an output node, input node or non of the two. Further, for BDDs, vp also includes the level of the nodes. op is an optimization algorithm for the given graph-based representation and ep are edge properties. For the example of BDDs, the edge properties denote if an edge is a *true* or a *false* edge, while for AIGs they denote if an edge is inverted or not.

B. Proposed Node Extraction Algorithm

In this section we present a novel algorithm for node extraction of a given graph-based representation.

Algorithm 1 Node Extraction

```
1: function NodeExtraction(CMOS,ReRAM,memFun,ep,vp)
2:   res = CMOS.node_count
3:   smallerGraphFound=false
4:   for all memFun in memFuncts do
5:     matches.push_back(SubgraphMatching(CMOS,memFun,ep))
6:   end for
7:   matchesByNodes=GroupMatchesByOuputNodes(matches)
8:   for all matches in CartProd(matchesByNodes) do
9:     NewCMOS = CMOS
10:    NewReRAM = ReRAM;
11:    for all match in matches do
12:      removedNodes=RemoveNodesFromGraph(NewCMOS,match)
13:      AddNodesToToGraph(NewReRAM,removedNodes)
14:    end for
15:    nodeCount=NodeExtraction(NewCMOS,NewReRAM,memFun,ep)
16:    if nodeCount<res then
17:      RetCMOS=NewCMOS
18:      RetReRAM=NewReRAM
19:      res=nodeCount
20:      smallerGraphFound=true
21:    end if
22:  end for
23:  if smallerGraphFound then
24:    CMOS=RetCMOS
25:    ReRAM=RetReRAM
26:  end if
27:  return res
28: end function
```

A recursive algorithm is depicted in Algorithm 1. The parameters are the graph which is to be implemented in CMOS (parameter *CMOS*), the graph which is to be implemented in ReRAM (parameter *ReRAM*), the set of ReRAM functions (parameter *memFun*) and the edge and vertex properties (parameters *ep* and *vp*). The parameter *CMOS* is initialized with the graph of the input function *f*, while the parameter *ReRAM* is initialized as an empty graph.

In Line 4-6 a modified subgraph matching for all given ReRAM functions is performed. It's purpose is to find all subgraphs in *f*, which match a given ReRAM function. Our implementation is based on Boost Graphs [18] implementation of the vf2 algorithm (an algorithm form finding of subgraph isomorphisms, see [19]. We do not detail it here due to page limitations) with tailored modifications. These modifications are:

- Subgraph matching is performed with respect to the edge properties (the edge properties must match exactly)
- Output nodes of the ReRAM function must match with output nodes
- Edges between terminal nodes are ignored
- For BDDs, if two nodes in a ReRAM function are on the same level, the matched nodes must be on the same level as well

Successively, in Line 7, the matches are sorted with respect to their output nodes, such that the variable $matchesByNodes$ is a set of vectors, where each vector contains pointers to all matches of subgraphs which start at the same output node. In Line 8-22 the Cartesian product of these

vectors is generated. Thus, the Cartesian product is a set of sets of matches, where each set of matches contains only matches which start at different output nodes. In Line 11-22 all entries of the Cartesian product are evaluated. First, in Line 12, the matched nodes are removed from the CMOS graph and added to the ReRAM graph in Line 13. Then, the algorithm is called recursively. Finally, if the final CMOS graph is smaller than the previously smallest computed graph, the new result is saved.

The algorithm moves as many nodes from the CMOS part to the ReRAM as possible and returns the corresponding graphs as output.

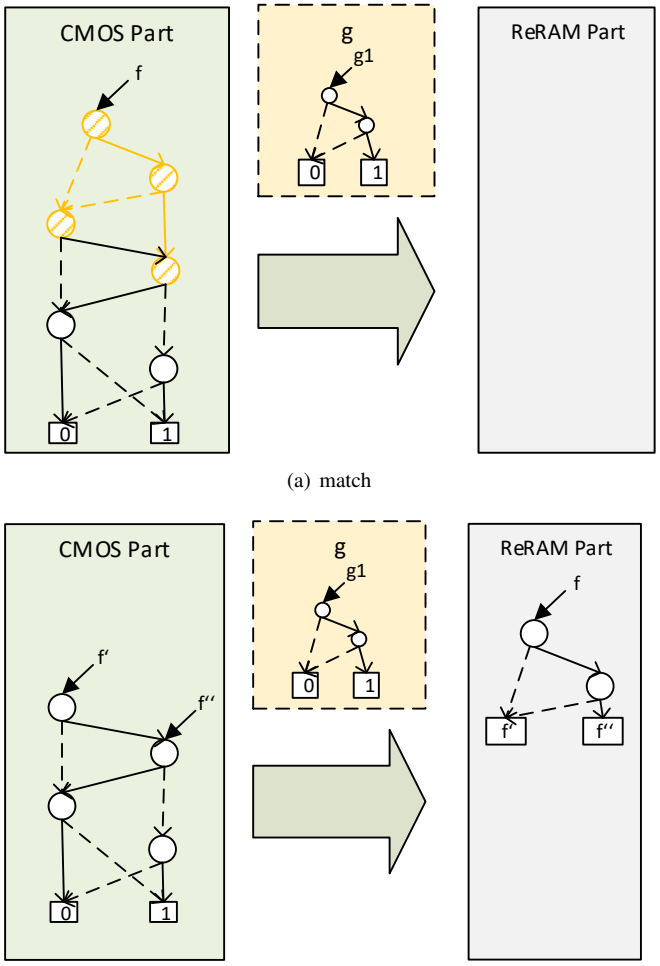
C. Design Space Exploration for BDDs

In order to minimize the size of the BDD representation of the CMOS part for a given function, we extend the method proposed in [5]. In [5], an *Evolutionary Algorithm* (EA) is used to optimize a given BDD with respect to different optimization objectives by finding an optimal variable ordering. The EA can be used to minimize the size of the BDD, by setting the size as the optimization objective. We have adjusted the EA by setting the size of the CMOS part of a given BDD as optimization objective. For each variable ordering which is evaluated by the EA, we generate the corresponding BDD and apply Algorithm 1. As edge properties, we pass for each edge, if it is a *true* or a *false* edge. Finally, the size of the BDD part, which is to be implemented in CMOS, is evaluated.

An example for node extraction applied to BDDs is given in Fig. 2. *true* edges are solid, while *false* edges are dashed. Fig. 2(a) shows the initial configuration, where no node extraction has been applied. The function f represented as BDD has to be implemented in CMOS completely (shown in the box on the left hand side) and ReRAM works as a regular memory without performing any computations (box on the right hand side). The set of ReRAM functions g consists of a single function g_1, which implements the AND-function shown in the dashed box above the arrow. When performing node extraction, we can see that there is a match between g_1 and the top of f. The output node of g_1 matches the output node of f, *true* edges match *true* edges and *false* edges match *false* edges. This is depicted as dashed yellow nodes and yellow edges. All nodes, which do not match terminal nodes of g_1 can be extracted to the ReRAM part. The nodes, which match terminal nodes of g_1 become the new output nodes of the CMOS part. The resulting graph is shown in Fig. 2(b). f' and f'' are the new outputs of the CMOS part. We use the outputs of the CMOS part as terminal nodes of the function in the ReRAM part (regardless if the initial graph representation was a BDD or an AIG).

D. Design Space Exploration for AIGs

In order to minimize the size of the CMOS part of a given AIG, we extend the algorithm *dc2* in ABC. *dc2* is a heuristic which uses a sequence of rewriting, refactoring and balancing steps, which allow to reduce the size of the graph without changing the implemented function. We apply Algorithm 1 at the end of dc2 in order to further reduce the size. As edge



(a) match

(b) result after node extraction

Fig. 2. Example node extraction

properties, we pass for each edge if it is inverted or not. Thus, node extraction for AIGs works similar as for BDDs. Instead of matching *true* and *false* edges, we match inverted and regular edges.

V. Experimental Evaluation

All experiments have been carried out on an Intel® Xeon® CPU E5-2630 v3 @ 2.40GHz with 64GB memory running Linux (Fedora release 22). We have used CUDD 3.0.0 [20] as a library for BDDs and ABC [21] for the construction of AIGs. For the evaluation we have used benchmarks from the well-known ISCAS89 benchmark-suite [22] with 13 to 54 input variables and 13 to 43 output variables and AND and OR as ReRAM functions.

The results of our experiments are shown in Table I. The first column shows the name of the benchmark. The second column denotes how many inputs and outputs the corresponding circuit has. The Columns 3-6 show the results for the experiments with BDD as graph representation, while the Columns 7-10 show the results for the experiments where AIGs have been employed as representation. In the following

TABLE I
SYNTHESIS RESULTS OF THE PROPOSED HYBRID APPROACH

Circuit		Size Reduction for BDDs				Size Reduction for AIGs			
Name	I/Os	#Nodes Init	#Nodes EA	#Nodes ReRAM (CMOS/ReRAM)	Reduction Rate	#Nodes Init	#Nodes dc2	#Nodes ReRAM (CMOS/ReRAM)	Reduction Rate
s208	19/10	1049	59	36/30	40.0%	79	52	48/4	7.7%
s298	17/20	132	87	85/2	2.3%	125	92	87/5	5.4%
s344	24/26	265	131	115/13	12.2%	189	136	135/1	0.7%
s349	24/26	265	124	122/5	1.6%	189	136	135/1	0.7%
s382	24/27	195	122	121/10	0.8%	154	115	110/5	4.3%
s386	13/13	285	114	107/8	6.1%	168	109	98/11	10.1%
s400	24/27	195	122	119/5	2.5%	154	115	110/5	4.3%
s420	35/18	262207	132	78/65	40.9%	167	104	TO	TO%
s444	24/27	236	121	121/0	0.0%	159	123	119/4	3.3%
s510	25/13	19096	157	157/0	0.0%	256	225	220/5	2.2%
s526	24/27	258	123	121/4	1.6%	200	144	140/4	2.8%
s641	54/43	1465	560	540/20	3.6%	479	358	346/12	3.4%
s820	23/24	2687	239	229/14	4.2%	351	253	TO	TO%
s832	23/24	2687	241	226/14	6.2%	351	251	TO	TO%
					avg: 8.7%				avg: 3.2%

subsections we describe the evaluation of the experiments for BDDs and AIGs.

A. Evaluation for BDDs

In order to evaluate our approach for BDDs, we have applied an *Evolutionary Algorithm* (EA) for the optimization of the BDD size as proposed in [5]. We have set the population size to three times the number of primary inputs, but no larger than 120. The algorithm terminates after 200 generations.

First, we have applied the EA to each circuit without node extraction in three independent runs and have evaluated the best out of them, i.e. the run which returned the smallest circuit size. Successively, we have adjusted the size evaluation such that the node extraction algorithm is applied to the BDD and the size of the CMOS part is evaluated. Again, we have evaluated the best out of three independent runs.

The results are shown in Columns 3-6 in Table I. Column 3 presents how many nodes the corresponding BDD has, if the natural variable order (i.e. $x_1 < x_2 < \ldots < x_n$) is used. Successively, in Column 4 we present the size of the BDD returned by the EA, if no node extraction is applied. Finally, Column 5 shows the results of the EA with node extraction. The first number represents the number of nodes in the graph for the CMOS part, while the second number is the size of the ReRAM part. Column 6 shows the reduction of nodes of the CMOS part for the result of the EA with node extraction compared to the result of the EA without node extraction.

It can be seen that the reduction of the graph size obtained by using node extraction varies a lot (ranging from 0% to 40.9%) with an average of 8.7%. The efficiency of the node extraction is strongly dependent on the choice of the given ReRAM functions and the structure of the BDD representation of the circuit. If the ReRAM functions match the structure of the BDD representation near the output nodes, node extraction can be used efficiently. However, if the structure of the BDD near the output nodes is different from that of the ReRAM functions, only little node extraction is possible.

B. Evaluation for AIGs

The open-source synthesis tool ABC [21] features the representation of circuits as AIGs. Since AIGs are not canonical, optimization is applied in order to reduce the size of AIGs. Optimization of AIGs with respect to their node count is implemented as a command in ABC called *dc2*. *dc2* is a heuristic algorithm, which is based on a sequence of rewriting, refactoring and balancing steps, which allow to reduce the size of the graph without changing the implemented function. In order to apply node extraction to a given AIG, we extend *dc2*. We apply Algorithm 1 to the given AIG at the end of *dc2* to further reduce the size. For the experiments we have applied *dc2* with node extraction and without node extraction to each benchmark.

The results are shown in Columns 7-10 in Table I. The columns for the AIG results are ordered in the same manner as for the BDD results.

It can be seen that the reduction of the CMOS part ranges from 0.0% (due to *timeout* (TO)) to 10.1% with an average of 3.2%. We had a TO for $s420$, $s820$ and $s832$ after 3 hours. Again, the efficiency of node extraction is very dependent on the ReRAM functions, however for AIGs the results are a lot more consistent than for BDDs. This is because the structure of the AIG representation of the used ReRAM functions is very simple (both only include a single non output non terminal node) and can be applied to all benchmark circuits to some degree. Eventhough node extraction results in TO for some benchmarks with AIG representation, while the algorithm terminated for all benchmarks using BDD-based representation, it could successfully reduce the complexity of *s444* and *s510* for which node extraction yielded no size reduction when a BDD-based representation is used.

C. ReRAM implementation

Following the design methodology of [6], we have calculated how many ReRAM devices and operations are needed to implement the ReRAM part of each benchmark, using

TABLE II
RERAM IMPLEMENTATION

Name	R_{BDD}	OP_{BDD}	R_{AIG}	OP_{AIG}
s208	58	54	15	8
s298	6	12	5	17
s344	42	12	3	3
s349	18	12	3	3
s382	33	24	10	10
s386	16	24	15	22
s400	17	30	10	10
s420	171	90	TO	TO
s444	NR	NR	10	7
s510	NR	NR	10	11
s526	15	24	5	14
s641	57	66	25	23
s820	48	24	TO	TO
s832	48	24	TO	TO

Eq. 1 and Eq. 2. The results are shown in Table II. The first column denotes the name of the benchmark. The second and third column show the needed number of ReRAM devices and operations for a BDD-based implementation, while the third and fourth column show how many ReRAM devices and operations are needed for an AIG-based implementation. NR means that node extraction did not result in any reduction in size of the CMOS part, while TO denotes a timeout.

We can see that the implementation of the ReRAM part using an AIG-based approach is smaller for all benchmarks than an BDD-based approach. Furthermore, less operations are needed for the AIG-based implementation for all benchmarks except for s298. Specially for s526 and s400, where the BDD-based and the AIG-based approach have extracted the same number of nodes it is remarkable that the AIG-based implementation needs a significant smaller number of ReRAM devices and operations compared to the BDD-based implementation. This leads to the conclusion that an AIG-based implementation is more suitable if the available ReRAM is small or low computation times are needed.

VI. CONCLUSION

In this paper we have presented a methodology for integrating ReRAMs into sequential circuits. To the best of our knowledge, we are the first to consider optimizing sequential circuits with ReRAM. We have proposed a novel node extraction algorithm for a given graph-based representation, which extracts nodes of the CMOS function to the ReRAM part, and have shown how to apply our algorithm to BDDs and AIGs. In the experiments we have applied our node extraction algorithm to a well known benchmark set and have shown that the size of the CMOS part can be reduced by an average of 8.7% for BDD-based optimization while for AIGs the reduction is about 3.2%. Furthermore, we have analyzed the complexity (number of needed ReRAM devices and operations) of the resulting ReRAM implementation.

REFERENCES

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

[2] 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 *Proceedings of IEEE International Symposium on Circuits and Systems*, 2010, pp. 1967–1970.

[3] Q. Xia, W. Robinett, M. W. Cumbie, N. Banerjee, T. J. Cardinali, J. J. Yang, W. Wu, X. Li, W. M. Tong, D. B. Strukov, G. S. Snider, G. Medeiros-Ribeiro, and R. S. Williams, "Memristor-cmos hybrid integrated circuits for reconfigurable logic," *Nano Letters*, vol. 9, no. 10, pp. 3640–3645, 2009. [Online]. Available: https://doi.org/10.1021/nl901874j

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

[5] 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.

[6] S. Shirinzadeh, M. Soeken, P. Gaillardon, and R. Drechsler, "Logic synthesis for rram-based in-memory computing," *IEEE Trans. on CAD of Integrated Circuits and Systems*, vol. 37, no. 7, pp. 1422–1435, 2018.

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

[8] 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.

[9] S. Shirinzadeh, M. Soeken, P. Gaillardon, and R. Drechsler, "Fast logic synthesis for rram-based in-memory computing using majority-inverter graphs," in *2016 Design, Automation & Test in Europe Conference & Exhibition, DATE 2016, Dresden, Germany, March 14-18, 2016*, 2016, pp. 948–953.

[10] 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.

[11] 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.

[12] 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.

[13] 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.

[14] D. Fried, L. M. Tabajara, and M. Y. Vardi, *BDD-Based Boolean Functional Synthesis*, ser. Lecture Notes in Computer Science. Springer, 2016, vol. 9780.

[15] N. Li and E. Dubrova, "Aig rewriting using 5-input cuts," in *Int'l Conf. on Comp. Design*, 2011, pp. 429–430.

[16] A. Chakraborty, R. Das, C. Bandopadhyay, and H. Rahaman, "Bdd based synthesis technique for design of high-speed memristor based circuits," in *2016 20th International Symposium on VLSI Design and Test (VDAT)*, 2016, pp. 1–6.

[17] R. E. Bryant, "Graph-based algorithms for boolean function manipulation," *IEEE Trans. on Comp.*, 1986.

[18] J. Siek, L. Lee, and A. Lumsdaine, *The Boost Graph Library: User Guide and Reference Manual*. Addison-Wesley, 2002.

[19] L. P. Cordella, P. Foggia, C. Sansone, and M. Vento, "A (sub)graph isomorphism algorithm for matching large graphs," *IEEE Transactions on Pattern Analysis and Machine Intelligence*, vol. 26, no. 10, pp. 1367–1372, 2004.

[20] F. Somenzi, "CUDD: CU Decision Diagram package-release 3.0.0," University of Colorado at Boulder, 2015.

[21] A. Mischenko, M. Case, R. Brayton, and S. Jang, "Scalable and scalably-verifiable sequential synthesis," in *International Conference on Computer-Aided Design*, 2008.

[22] F. Brglez, D. Bryan, and K. Kozminski, "Combinational profiles of sequential benchmark circuits," in *IEEE International Symposium on Circuits and Systems*, 1989, pp. 1929–1934 vol.3.