Preserving Self-Duality During Logic Synthesis for Emerging Reconfigurable Nanotechnologies

Shubham Rai*, Heinz Riener†, Giovanni De Micheli†, Akash Kumar*
*CfAED Technische Universität Dresden, Germany
†Integrated Systems Laboratory, EPFL, Lausanne, Switzerland

***Abstract*—Emerging reconfigurable nanotechnologies allow the implementation of self-dual functions with a fewer number of transistors as compared to traditional CMOS technologies. To achieve better area results for *Reconfigurable Field-Effect Transistors* (RFET)-based circuits, a large portion of a logic representation must be mapped to self-dual logic gates. This, in turn, depends upon how self-duality is preserved in the logic representation during logic optimization and technology mapping. In the present work, we develop Boolean size-optimization methods–a rewriting and a resubstitution algorithm using *Xor-Majority Graphs* (XMGs) as a logic representation aiming at better preserving self-duality during logic optimization. XMGs are more compact for both unate and binate logic functions as compared to conventional logic representations such as *And-Inverter Graphs* (AIGs) or *Majority-Inverter Graphs* (MIGs). We evaluate the proposed algorithm over crafted benchmarks (with various levels of self-duality) and cryptographic benchmarks. For cryptographic benchmarks with a high self-duality ratio, the XMG-based logic optimisation flow can achieve an area reduction of up to 17% when compared to AIG-based optimization flows implemented in the academic logic synthesis tool ABC.**

I. Introduction

Ambipolar nanotechnologies form a class of emerging nanotechnologies featuring both n- and p-type functionality from a single transistor. Recently, it has been demonstrated, that Boolean functions which are *self-dual*, can be implemented efficiently using *Reconfigurable Field-Effect Transistors* (RFETs) [20]. In the present work, we explore logic synthesis methods which can specifically take advantage of the self-duality of Boolean functions to achieve area improvements.

Within design automation, logic synthesis is an integral part which optimizes a logic network in terms of a cost function, typically focusing on the reduction of area or delay. Recently, novel multi-level logic representations such as *Xor-And Graphs* [11] or *Xor-Majority Graphs* (XMGs) [8, 6] have been proposed, which enrich conventional *And-Inverter Graphs* (AIGs) [3] and *Majority-Inverter Graphs* (MIGs) [2] with an additional Xor primitive. These new logic representations offer more compactness and enable better runtimes for logic optimization and minimization flows [8, 23]. Particularly in the context of RFET-based circuits, the logic primitives used in XMGs— *Majority* and *Xor* gates, can better preserve self-duality as both, the majority-of-three and the odd-input Xor function, are self-dual.

In the present work, we explore the usage of XMGs in logic optimization methods to achieve area reduction after technology mapping for RFET-based circuits. The two contributions are summarized as follows:

1) Conventional logic representations such as AIGs or MIGs suppress self-duality during logic optimization. We propose an XMG-based logic synthesis flow that allows preserving the self-duality during logic optimization. This flow enables better area reductions after technology mapping for RFET-based circuits in comparison to state-of-the-art logic optimization flows. In an experimental evaluation over crafted benchmarks with varying levels of self-duality, we achieve area reductions of up to 6.59%, 3.24%, 3.63%, and 5.95% as compared to state-of-the-art logic optimization flows *rewrite;resub (until convergence)*, *compress2rs*, *dc2* and *dch*, respectively. Over cryptographic benchmarks, using the proposed XMG-based flow, area reductions up to 17% are achieved for benchmarks with high self-duality.
2) We propose state-of-the-art resubstitution and rewriting algorithms for XMGs. Our resubstitution algorithm uses a new filtering rule for 3-input Xor gates (Xor3). This filtering rule reduces the average runtime for resubstitution over the EPFL benchmarks by 46.13% while preserving the quality. Our rewriting algorithm for XMGs, called *exact XMG rewriting*, uses cut enumeration, NPN canonization, and exact synthesis. In contrast to the previous XMG rewriting approaches, the algorithm uses structural hashing to utilize existing logic and can achieve size reduction even if a smaller subnetwork is replaced with a larger one.

II. Background

A. Reconfigurable nanotechnologies

Ambipolarity, as a phenomenon is observed at lower technology nodes, but often suppressed using process techniques [13]. The class of emerging nanotechnologies which aims to exploit this ambipolarity is often termed as emerging reconfigurable nanotechnology and the devices are called *reconfigurable field-effect transistors* (RFETs). These devices demonstrate both n- and p-type functionality from a single device on application of an external bias. Multiple device geometries based on various materials like silicon [9, 13], germanium [28], carbon [12] etc. have been proposed which exhibit near to full electrical symmetry in both n- and p-type functionality. This electrical symmetry is shown as V-shaped curve in Fig. 1. Logic gates based on these devices are able to exhibit more than one functionalities [17, 19, 18] as shown in Fig. 2.

B. Self-dual functions

A logic function $f(x_1, \ldots, x_n)$ is said to be self-dual [26] if

$$f(x_1, \ldots, x_n) = \bar{f}(\bar{x}_1, \ldots, \bar{x}_n) \tag{1}$$

By complementing the function, an equivalent self-dual formulation is $\bar{f}(x_1, \ldots, x_n) = f(\bar{x}_1, \ldots, \bar{x}_n)$. For a particular

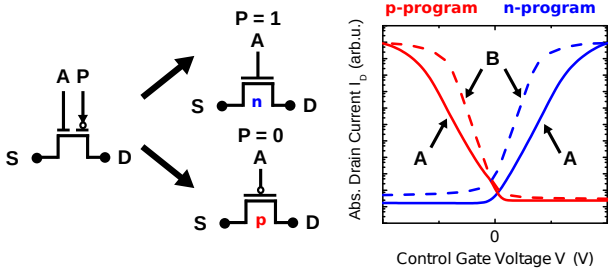
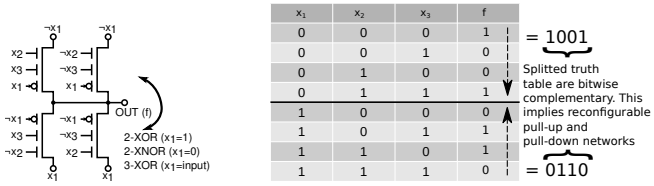


Fig. 1: A generic RFET showing two gate terminals: The program (signal P) and the control gate (signal A) [19]. The adjacent curve shows the V-shaped curve representing electrical symmetry for n- and p-type functionality.



(a) (b)

Fig. 2: (a) Reconfigurable logic gate Xor3 demonstrated in [9, 19]. It shows how functionality changes with value of P. (b) Truth-Table for Xor3 logic gate. The truth-table is split over the value of x_1 which is the reconfigurable input.

instance of $x_1, \ldots, x_n$, $f(x_1, \ldots, x_n)$ and $\bar{f}(\bar{x}_1, \ldots, \bar{x}_n)$ are *dual* to each other.

Fig. 2b shows an RFET-based 3-input *Xor* function which is a self-dual function. Here, when the truth-table is divided over the value of x_1 (or any other arbitrary literal), the two halves of the *Xor3* truth-table (*Xor2* and *XNor2* functions) are dual to each other.

In [20], the authors showed that self-dual functions are the logical abstraction for reconfigurable nanotechnology. The multiple functionality exhibited by RFET-based logic gates are due to the interchangeable pull-up and pull-down networks as shown in Fig. 2. The switching of polarities of individual transistors in their respective pull-up and pull-down networks is caused by changing the potential at the program gate as shown in Fig. 2a. This change in potential at the program gate causes the PFET to become NFET and vice-versa causing the pull-up and pull-down networks to flip as shown in Fig. 2. This corresponding switch in electrical behavior is abstracted conveniently in a self-dual function as shown by two parts of the truth-table of Fig. 2b. The two parts of the truth-table represent the interchangeable pull-up and pull-down networks.

C. Terminology

We introduce some terminologies here, which will be used through the rest of the paper.

1) *Density of self-duality*: We define the term density of self-duality for a circuit or logic network as the ratio of total number of self-dual nodes to the total number of nodes.

2) *Trivial and non-trivial self-dual functions*: Self-dual functions are fewer (square root of total number of functions) as compared to non-self-dual functions [25]. Moreover, among two-input functions, self-duality exists in those functions which are equivalent to either of the inputs or to their complements (for example, $f(a, b) = f(a)$). Such functions are implemented in circuits as wires (or use a single inverter) and hence their implementation requires very few transistors. Thus, two-input self-dual functions are termed here as *trivial functions* as they have little impact on the overall area of the circuit.

For functions with more than two inputs, their implementation with RFETs require fewer number of transistors as compared to their CMOS counterpart. These functions will have a direct impact on the area of the circuit. Hence, 3 or more input functions which are self-dual are termed as *non-trivial functions*.

D. Earlier works on XMG

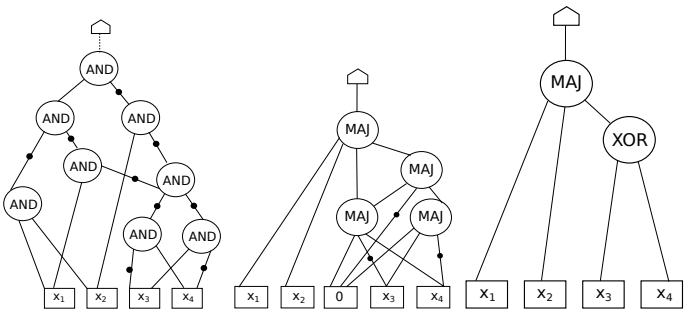
Xor-Majority Graphs in their current format, were first introduced in [8] as a means for underlying logic representation for exact synthesis. As exact synthesis uses SAT solving or enumeration, its runtime directly depends upon the size of the logic representation. Since XMGs have both binate (*Xor*) as well as unate (*Maj*) nodes, it gives a size-proportional logic representation for both n-input unate and binate functions as compared to the poor representation of binate logic (*Xor*-based logic) by MIGs or AIGs [6]. Algebraic optimizations for XMGs were proposed in [6]. They explored Boolean algebraic optimizations for *Xor* and *Xor-Maj* logic and were able to achieve depth optimizations.

In the present work, we explore logic synthesis in order to maximize the self-duality within a circuit so that they can be efficiently mapped to RFETs. Since our objective is primarily post-mapping area, we are focusing on size optimization and hence, have not considered algebraic optimizations in the present work.

III. Motivation

Due to their device-level reconfigurability, RFETs allow efficient implementations of non-trivial self-dual logic functions in terms of the number of transistors [20] as compared to CMOS. For example, a 3-input Xor logic gate (shown in Fig. 2) needs $4 + 2$ (for P and P′) transistors when realized with RFETs as compared to 22 transistors when realized in the CMOS technology [19]. This implies that circuit implementations with RFETs lead to area reductions, if they have a higher density of non-trivial (3 or more input functions) self-dual gates. Hence, to take advantage of this property, it is imperative that self-duality in a logic representation is preserved by logic optimizations. From logic representation perspective, if we consider AIGs (consisting of two-input And gates with complement-edge attributes), a non-trivial self-dual function will be decomposed into multiple AIG nodes and, hence, during logic optimization, self-duality may be lost. Similarly for MIG, parity-based self-dual functions cannot be represented in a compact manner using *Maj* nodes alone, which can again result in a loss of self-duality during logic optimization [6].

In contrast, XMGs use *Xor* and *Maj* nodes as logic primitives. Every Majority and odd-input Xor function is self-dual, such that using XMGs can better preserve self-duality during logic optimization as compared to other logic representations. This



(a) AIG (b) MIG (c) XMG

Fig. 3: Different logic representations for the function, $f = \langle x_1, x_2, (x_3 \oplus x_4)\rangle$. One can notice that the number of nodes and edges are the least in case of XMGs. The dot represents complemented edges.

can be easily seen in Fig. 3. The figure shows three logic representations of the same function $f = \langle x_1, x_2, (x_3 \oplus x_4)\rangle$. The logic representation as AIG requires 8 nodes, while the same function has 4 and 2 nodes when represented as MIG and XMG respectively. In the example, the number of edges in AIG or MIG representation is also much higher as compared to XMG representation. This leads to an increase in the number of competing structural cuts of the logic network in logic optimization and technology mapping phase. Keeping in mind the above advantages, we develop an XMG-based logic optimization flow which addresses these issues and helps to achieve area reductions for RFET-based circuits.

IV. Logic synthesis techniques

While the earlier works [8, 6] introduced the basic logic optimizations using XMGs, we extend the support of XMGs with advanced Boolean resubstitution method as well as NPN-based cut-rewriting techniques.

A. Boolean XMG resubstitution and filtering

Boolean resubstitution is a logic optimization method that re-expresses the function of a node n in a logic network N using nodes, called *divisors*, already present in N. Nodes that are exclusively used by n and are not required by any other logic in the logic network become free and can be removed. A resubstitution leads to a size reduction if the number k of newly added nodes to re-express a node's function is less than the number l of removed nodes in its *maximum fanout-free cone* (MFFC, [15]).

Resubstitution algorithms are available for different multi-level logic representations including AIGs [14, 15], MIGs [24, 23], and logic networks [16]. Computing three-input Xor (Xor3 is a self-dual logic gate) resubstitutions is particularly time-consuming because divisor filtering techniques developed for And and Or operations cannot be applied. To substitute a node n in a network with logic function $f_n(x)$ by a three-input Xor operation, three divisor nodes d_1, d_2, and d_3 have to be found, such that

$$f_n(x) = f_{d_1}(x) \oplus f_{d_2}(x) \oplus f_{d_3}(x) \tag{2}$$

for all assignments to the primary inputs x, where f_{d_1}, f_{d_2}, f_{d_3} are the divisor functions, respectively.

Algorithm 1: Boolean filtering and resubstitution

Data: Window W in a logic network with root node n
Result: Node resubstitute for n or $\perp$ if no resubstitution has been found

```
1  Set M ← W.computeMFFC(n);
2  Set D ← W.collectDivisors(n)\M;
3  Set TT ← W.simulate();
4  sortByDBP(D, TT, n);
5  for i ← 0 to |D| do
6      if 3 · DBP(D[i]) < DBP(n) then
7          return ⊥;
8      for j ← i + 1 to |D| do
9          if DBP(D[i]) + 2 · DBP(D[j]) < DBP(n) then
10             break;
11         for k ← j + 1 to |D| do
12             if f = TT[i] ⊕ TT[j] ⊕ TT[k] then
13                 return W.xor3_resub(n, D[i], D[j], D[k]);
14             if f = ¬TT[i] ⊕ TT[j] ⊕ TT[k] then
15                 return W.xor3_resub(n, D[i], D[j], D[k]);
16 return ⊥;
```

State-of-the-art Boolean resubstitution algorithms over-approximate the node functions using windowing to apply scalable truth-table computations. The algorithms have to iterate over all triples of nodes in a window of a root node n (excluding the root node's MFFC) to test if Eq. 2 holds. The first substitution possible that reduces the network's size is accepted. In the worst case, if no resubstitution can be accepted, $\mathcal{O}(w^3)$ tests are required for a window with w nodes.

Filtering techniques help to reduce the number of tests required and significantly speed-up the performance of resubstitution algorithms in practice. We develop a new filtering rule for three-input Xors guiding the search for divisors using distinguishing bit-pairs [4]: a resubstitution of a target node n with function $f(x)$ and divisors nodes d_1, d_2, d_3 with functions $f_{d_1}(x), f_{d_2}(x), f_{d_3}(x)$ over common window inputs x exists if and only if for any pair $\hat{x}_i \neq \hat{x}_j$ of input assignments

$$f(\hat{x}_i) \neq f(\hat{x}_j) \implies \bigvee_{1 \leq a,b \leq 3, a \neq b} d_a(\hat{x}_i) \neq d_b(\hat{x}_j). \tag{3}$$

Utilizing Eq. 3, we sort all divisor nodes in a window by the number of bit-pairs distinguished by the divisor with respect to the root node's target function. We define the *absolute distinguish bit power* $\mathrm{DBP}(n)$ of the root node n as the number of pairs $(\hat{x}_i, \hat{x}_j)$ of input assignments for which $f_n(x_i) \neq f_n(x_j)$, and we define the *relative distinguishing bit power* $\mathrm{DBP}_n(d)$ of a divisor d as the number of pairs $(\hat{x}_i, \hat{x}_j)$ of inputs assignments for which $f_n(\hat{x}_i) \neq f_n(\hat{x}_j)$ and $f_d(\hat{x}_i) \neq f_d(\hat{x}_j)$.

Algorithm 1 shows our Boolean filtering and resubstitution algorithm as pseudo code. The divisors are sorted (line 4) by their relative distinguishing bit power—higher relative distinguishing bit power will more likely lead to a possible resubstitution. We further leverage the relative distinguishing bit power to filter *insufficient* divisor triples. Given a sorted list $D = d_1, ..., d_w$ of divisors such that $\mathrm{DBP}_n(d_i) \geq \mathrm{DBP}_n(d_j)$ for all $i < j$, a single divisor d can never be completed to divisor triple that passes the test in Eq. 2 if $3 \cdot \mathrm{DBP}_n(d) < \mathrm{DBP}(n)$ (line 6). Since the list is sorted, no remaining divisor will pass this test either such that the algorithm can terminate (line 7).

For a similar reason, no divisor pair d_i, d_j, $i < j$, can be completed to a divisor triple that passes the test in Eq. 2 if $\mathrm{DBP}_n(d_i)+2\cdot\mathrm{DBP}_n(d_j) < \mathrm{DBP}(n)$ (line 9). In this case, the algorithm can proceed by selecting another candidate divisor d_i (line 10).

B. Exact XMG rewriting

Boolean rewriting is a logic optimization method that selects small parts of a logic network and replaces them with more compact implementations to reduce its number of nodes. State-of-the-art rewriting algorithms either rely on a database of pre-computed size-optimum subnetworks for all Boolean functions up to 5 inputs [15] or compute size-optimum subnetworks on-the-fly using exact synthesis [22, 21]. DAG-aware rewriting [15], fast cut enumeration techniques [7], NPN canonization [10] of Boolean functions, and efficient caching [21] enable scalability.

Rewriting XMGs has been first proposed in [8] using a two-step approach: (1) A logic network is mapped into a network of k-feasible lookup-tables (LUTs); (2) the k-feasible LUTs are resynthesized into size-optimum XMGs. By repeating the two steps until convergence, substantial size reduction can be achieved.

We propose an improved XMG rewriting approach, called *exact XMG rewriting*, that integrates both steps into one algorithm. For each node, in the logic network, the set of all k-feasible cuts is enumerated, each cut is simulated to obtain its Boolean functions, and the functions are resynthesized using exact synthesis. In contrast to the previous approach, our algorithm takes advantage of structural hashing to utilize the existing logic within the network, such that a global size reduction can be achieved even if a locally smaller subnetwork is replaced with a larger subnetwork. The algorithm can be parameterized with a set of gate primitives and supports synthesis of multiple candidates per cut function. A conflict limit in exact synthesis allows to limit the maximum synthesis effort per function.

V. Creating Self-dual Benchmarks

In order to evaluate the efficacy of our approach as compared to the state-of-the-art logic synthesis approaches, for RFET-based standard cell mapping, we generate benchmarks with varying density of self-dual logic gates within the circuit. We propose a metric called a *self-duality index* to vary the density of self-duality within a circuit. The rationale behind this is that if we have a larger number of self-dual components in the circuit, then more self-dual logic gates can be utilized during mapping. This leads to an improved area results for RFET-based circuit.

We generate multiple benchmarks using Algorithm 2. We start with an empty logic network and take four parameters as inputs– *number of Primary Inputs (PIs)* (num_pis), *number of levels* ($levels$), *number of nodes per level* ($nodes\_per\_level$) and *self-duality index* ($sdIndex$, whose value has to be between 1 and 10). Depending upon the value of *self-duality index*, for every 10 nodes added in the logic network, number of self-dual nodes added, is equal to *self-duality index* (line 9) and the remaining ($10-$ *self-duality index*) (line 11) nodes are normal nodes. By normal nodes, we mean adding logic nodes using AND, OR, XOR or constants while self-dual nodes implies adding Majority or 3-input XOR logic nodes. We maintain a signal list SL where we keep adding all the newly created nodes (line 12). We then randomly select nodes from the signal list SL to add to the circuit (line 7). The code for generating crafted benchmarks and the generated benchmarks are uploaded at https://github.com/shubhamrai26/iwls2020_experiments

Algorithm 2: Populating benchmarks with varying level of self-duality

```
   Data: num_pis, levels, nodes_per_level, sdIndex
   Result: XMG network N
1  Set signalList ← [];
2  Set sd_or_normal ← 0;
3  for k ← 0 to num_pis do
4  |  signalList.add(N.create_pi());
5  for i ← 0 to levels do
6  |  for j ← 0 to nodes_per_levels do
7  |  |  fanins ← signalList.randSubSet();
8  |  |  if sd_or_normal < sdIndex then
9  |  |  |  node ← N.create_selfdual_gate(fanins);
10 |  |  else
11 |  |  |  node ← N.create_normal_gate(fanins);
12 |  |  signalList.add(node);
13 |  |  sd_or_normal ← (sd_or_normal + 1)
   |  |   mod 10 ;
14 for o ∈ signalList.not_used() do
15 |  N.create_po(o);
16 return N
```

VI. Experiments and Discussion

In this section, we evaluate our approach with various experiments. All the implementations are integrated in the logic network library *mockturtle* from the EPFL logic synthesis libraries [27]. We extend the XMG network with the proposed Boolean methods such as XMG resubstitution and exact XMG rewriting.

A. Methodology

We first evaluate the speedup in runtime due to the filtering rule integrated in the XMG resubstitution algorithm. We then apply our XMG-based flow to the crafted benchmarks and the cryptography benchmarks. To produce area results, we use the standard mapper of the academic logic synthesis tool, ABC using the logic gates proposed in [19]. Further, we compare our XMG-based flow with the state-of-the-art synthesis scripts *rewrite and resub*, *compress2rs*, *dc2*, and *dch* implemented in ABC.

B. Runtime improvement with filtering rule

In order to measure the runtime improvement of the Xor3-based filtering rule, we run XMG resubstitution (with and without filtering) once on all arithmetic EPFL benchmarks. The third and the forth column in TABLE I show the runtime of the XMG resubstitution algorithm for both runs of the resubstitution command. On average, a runtime improvement of 46% is achieved with an average size penalty of $\sim 2\%$.

TABLE I: Runtime improvement using our filtering rule

Benchmark	Runtime without filter (in sec)	Runtime with filter (in sec)	Improvement %
adder	0.31	0.21	32.26
bar	2.34	0.90	61.54
div	58.85	35.58	39.54
hyp	391.29	321.19	17.92
log2	28.38	16.26	42.71
max	1.39	0.89	35.97
multiplier	33.53	13.44	59.92
sin	7.49	3.34	55.41
sqrt	29.77	13.64	54.18
square	17.77	6.78	61.85
Average			**46.13**

TABLE II: Comparison of final area for crafted benchmarks with respect to ABC scripts of *rw;rs*, *compress2rs*, *dc2* and *dch*

Sd-index	sd-ratio	init_area (geomean)	xmg-rwrs	xmg-c2rs (%)	xmg-dc2 (%)	xmg-dch (%)
1	36.14	662953.00	-3.59	0.10	-1.72	-3.74
2	40.23	658231.50	-1.18	1.83	0.16	-1.13
3	44.24	654716.00	-5.91	2.50	0.78	0.15
4	48.82	651544.50	1.17	3.09	1.59	1.05
5	53.41	645854.00	1.92	3.20	2.01	1.73
6	58.52	640258.50	2.62	3.22	2.34	2.32
7	64.53	636013.50	3.40	3.24	2.68	3.17
8	71.37	631824.00	4.02	3.00	2.94	3.67
9	79.35	625668.00	4.95	2.78	3.18	4.48
10	100.00	614186.00	6.95	1.66	3.63	5.95

C. Crafted self-dual benchmarks

Within this experiment, we use Algorithm 2 to populate multiple benchmarks with varying numbers of PIs, numbers of levels and numbers of nodes per levels. The self-duality index is taken care by the variable *sdIndex* whose value is iterated from 1 to 10 to populate 10 benchmarks for a single set of parameters. We apply exact XMG rewriting and resubstitution over XMG till convergence, and then carry out standard-cell mapping using RFET-centric generic library [19]. For comparison, we first compare our XMG-based flow with a similar ABC script of of *rewrite;resub* until convergence and then with other state-of-the-art scripts *compress2rs*, *dc2* and *dch* implemented in ABC. Since our approach converges after one iteration, it is fair enough to compare with the above three scripts from ABC.

TABLE II shows the comparison of post-mapping area carried out using different scripts for our crafted benchmarks. The first column shows the value of sd-index which signifies how many self-dual nodes have been added for every 10 nodes. The next column (sd-ratio) is the density of self-duality for the RFET-based circuit after optimization using Boolean methods proposed in Section IV. This is followed by a column of geometric mean of the initial area for the variants of the benchmark generated. Finally, the columns *xmg-rwrs*, *xmg-c2rs*, *xmg-dc2*, and *xmg-dch* show how the final area using XMG-based optimization compares with the ABC logic optimization flows. The numbers are in percentage where a positive value means that XMG gives better numbers as compared to the ABC scripts and vice-versa. For XMG-based flow one can notice a direct correlation between the improvements in area and the higher values for self-duality ratios. As compared to the ABC flow of *rewrite;resub* (until convergence) which directly compares with our flow, we can notice that as the self-duality increases, the improvement also increases. Among other powerful optimization ABC scripts, *compress2rs* script gives the closest result across the sd-index. Hence, this experiment shows that with increase in the sd-index, XMG based optimization gives better numbers as compared to the state-of-the-art ABC scripts.

D. Cryptography protocol benchmarks

While the previous experiment was conducted on crafted benchmarks, we now evaluate our approach on cryptography benchmarks to establish our conjecture that an increase in self-duality within an RFET-based circuit can be better optimized with XMGs. These benchmarks were taken from high-level cryptography protocols such as *Fully Homomorphic Encryption* (FHE) and secure *Multy-Party Communication* (MPC) [1][1]. The benchmark suite contains circuits ranging from block ciphers (AES and DES) and hash functions such as (MDA-5 and SHA) to arithmetic functions (adders and comparators).

The results are shown in TABLE III. As in the case of earlier benchmarks, here also we compare the post-mapping area. The first column shows the sd-ratio after carrying out optimization using algorithms as mentioned in Section IV. The column *xmg_area* shows the area using our XMG-based approach.

E. Discussion

We can see that our XMG-based approach achieves better area numbers in case of benchmarks with higher self-duality as compared to ABC scripts. This can be ascertained to the fact that most of the benchmarks from the cryptography domain have a high density of self-dual gates. They also have high density of parity functions, as parity functions are integral logic functions in any cryptographic applications. For the benchmarks, *md5*, *SHA-1* and *SHA-256*, our XMG-based approach outperforms other ABC-based scripts. This is also coherent with their high sd-ratio values. For small benchmarks, such as *adder*, all flows reach the optimal area results.

Certain interesting observations can be made here—firstly, circuits such as *AES* which has a low density of self-duality returns better area with our approach. This is due to the fact that it has a high parity function density which is better optimised by XMG as compared to other flows. Secondly, in a one-to-one comparison, our XMG-based rewriting and resubstitution techniques lead to more powerful optimization as compared to ABC *rewrite; resub* scripts. This is apparent as our XMG-based approach achieves upto 17% (in case of *md5*) better area. Another observation is that for benchmarks such as *comparator*, our XMG-based approach returns inferior results. We noticed here, that some of the cuts were not mapped optimally to logic gates (or combinations of logic gates). The reason can be ascertained to the fact that, since ABC technology mapper is used, it decomposes three-input Majority primitives and Xor primitives to multiple two-input And primitives as is evident from Fig. 3. During technology mapping, ABC carries out cut-enumeration which in case of the XMG-based approach, leads to thrice the number of competing cuts as compared to technology mapping after logic optimization script (*compress2rs*, *dc2* and *dch*) within ABC.

To investigate the above observation, we started with a smaller subcircuit, where the mapping for the XMG-based approach

[1]The benchmarks were obtained from https://homes.esat.kuleuven.be/nsmart/MPC/

TABLE III: Comparison of mapped area for the cryptography benchmarks using XMG optimization as compared to ABC scripts

Benchmarks	**sd-ratio**	**init_area**	**c2rs_area**	**dc2_area**	**dch_area**	**rwrs_area**	**ABC best**	**xmg_area**	**impr best**	**impr rwrs**
AES-expanded_256	17.59	94661.50	85270.50	89073.00	88622.00	88988.50	85270.50	**84342.50**	**1.09**	**5.22**
AES-non-expanded_1536	17.09	118531.50	104173.00	111344.00	110196.00	110702.00	104173.00	104338.50	-0.16	**5.75**
DES-expanded_128	35.62	42437.50	39912.50	41682.50	41454.50	39417.50	39417.50	45266.50	-14.84	-14.84
DES-non-expanded_832	36.07	42718.50	40633.50	41711.50	41395.50	39397.00	39397.00	45198.00	-14.72	-14.72
adder_32bit	85.29	270.00	270.00	270.00	270.00	270.00	270.00	**270.00**	0.00	0.00
adder_64bit	81.43	542.00	542.00	542.00	542.00	542.00	542.00	**542.00**	0.00	0.00
adder_128bit	72.30	1666.00	1086.00	1086.00	1086.00	1086.00	1086.00	**1086.00**	0.00	0.00
comparator_32bit_signed_lt	44.86	295.00	244.00	243.50	241.00	242.50	241.00	302.50	-25.52	-24.74
md5	47.86	113198.50	113451.50	115374.00	115377.50	120937.50	113198.50	**100380.50**	**11.32**	**17.00**
mult_32x32	40.62	10964.00	10885.00	9632.50	10141.50	11643.00	9632.50	12447.50	-29.22	-6.91
sha-1	60.07	161321.50	158645.00	165201.00	166241.00	172729.00	158645.00	**146005.50**	**7.97**	**15.47**
sha-256	69.44	274680.00	285680.50	281488.00	280742.00	282895.00	274680.00	**254327.00**	**7.41**	**10.10**

requires additional Minority gates. Although the two code snippets are functionally equivalent, the mapping result of the first code snippet is more than the second by 1 unit. Such mapping leads to poor area results for XMG-based approach. This is particularly more pronounced in cases, where circuits have very low density of parity logic.

```
/* Code snippet 1 */
assign w1 = x2 & ~x4;
assign w2 = (x1 & x3) | (x1 & ~w1) |
    (x3 & ~w1);
assign w3 = w1 & w2;
/* Code snippet 2 */
assign w1 = x1 & x2 & x3 & ~x4;
```

The reason behind the above finding is that, the two code snippets have different subject graphs and the mapping within ABC can suffer from structural bias [5]. We also applied our approach to EPFL benchmarks and found out that EPFL benchmarks have low self-duality density and hence they are not very representative of our approach. Additionally, the above investigation is seen in various benchmarks (for example—*arbiter* and *bar*). One of the possible solutions to mitigate the above issue is to design an XMG-based technology mapper which does not decompose individual XMG nodes to smaller primitives. However, development of such a technology mapper is beyond the scope of this work.

VII. Summary and Conclusion

This work explores logic synthesis from an emerging nanotechnology perspective. Keeping in mind that self-dual logic functions are implemented efficiently with RFETs, we have explored XMGs as logic representation to exploit self-duality in circuits because (i) they provide a compact logic representation and (ii) majority and odd-input parity functions are self-dual and can efficiently be represented by XMGs. We have developed advanced Boolean methods such as resubstitution and rewriting techniques for XMGs to enable powerful optimizations. We have shown that circuits with a high density of self-duality achieve better area results for XMG-based approaches as compared to the state of the art.

Acknowledgments

This research was supported in part by the German Research Foundation (DFG), project SecuReFET (Project Number: 439891087), and by the EPFL Open Science Fund.

References

[1] M. R. Albrecht et al. “Ciphers for MPC and FHE”. In: *EUROCRYPT*. Springer. 2015.
[2] L. Amarú, P. E. Gaillardon, and G. De Micheli. “Majority-Inverter Graph: A New Paradigm for Logic Optimization”. In: *TCAD* (2016).
[3] R. K. Brayton and A. Mishchenko. “ABC: An Academic Industrial-Strength Verification Tool”. In: *CAV*. 2010.
[4] K. Chang, I. L. Markov, and V. Bertacco. “Fixing Design Errors With Counterexamples and Resynthesis”. In: *TCAD* (2008).
[5] S. Chatterjee. *On algorithms for technology mapping*. University of California, Berkeley, 2007.
[6] Z. Chu et al. “Structural Rewriting in XOR-Majority Graphs”. In: *ASPDAC*. 2019.
[7] J. Cong, C. Wu, and Y. Ding. “Cut Ranking and Pruning: Enabling a General and Efficient FPGA Mapping Solution”. In: *ISFPGA*. 1999.
[8] W. Haaswijk et al. “A novel basis for logic rewriting”. In: *ASP-DAC*. 2017.
[9] A. Heinzig et al. “Reconfigurable silicon nanowire transistors”. In: *Nano Letters* (2012).
[10] Z. Huang et al. “Fast Boolean matching based on NPN classification”. In: *FPT*. 2013.
[11] I. Háleček, P. Fišer, and J. Schmidt. “Are XORs in logic synthesis really necessary?” In: *DDECS*. 2017.
[12] Y. Lin et al. “High-performance Carbon Nanotube Field-effect Transistor with Tunable Polarities”. In: *TNano.* (2005).
[13] M. De Marchi et al. “Polarity control in double-gate, gate-all-around vertically stacked silicon nanowire FETs”. In: *IEDM*. 2012.
[14] A. Mishchenko and R. K. Brayton. “Scalable logic synthesis using a simple circuit structure”. In: *IWLS*. 2006.
[15] A. Mishchenko, S. Chatterjee, and R. K. Brayton. “DAG-aware AIG rewriting a fresh look at combinational logic synthesis”. In: *DAC*. 2006.
[16] A. Mishchenko et al. “Scalable don’t-care-based logic optimization and resynthesis”. In: *ACM TRECTS.* (2011).
[17] J. Nevoral, R. Růžička, and V. Šimek. “From Ambipolarity to Multi-functionality: Novel Library of Polymorphic Gates Using Double-Gate FETs”. In: *DSD*. 2018.
[18] S. Rai, M. Raitza, and A. Kumar. “Technology mapping flow for emerging reconfigurable silicon nanowire transistors”. In: *DATE*. 2018.
[19] S. Rai et al. “Designing Efficient Circuits Based on Runtime-Reconfigurable Field-Effect Transistors”. In: *TVLSI* (2019).
[20] S. Rai et al. “DiSCERN: Distilling Standard-Cells for Emerging Reconfigurable Nanotechnologies”. In: *DATE*. 2020.
[21] H. Riener, A. Mishchenko, and M. Soeken. “Exact DAG-aware Rewriting”. In: *DATE*. 2020.
[22] H. Riener et al. “On-the-fly and DAG-aware: Rewriting Boolean Networks with Exact Synthesis”. In: *DATE*. 2019.
[23] H. Riener et al. “Scalable Generic Logic Synthesis: One Approach to Rule Them All”. In: *DAC*. 2019.
[24] H. Riener et al. “Size Optimization of MIGs with an Application to QCA and STMG Technologies”. In: *NANOARCH*. 2018.
[25] T. Sasao. “History of Switching Theory”. In: *Switching Theory for Logic Synthesis*. Springer, 1999.
[26] T. Sasao. *Switching theory for logic synthesis*. Springer Science & Business Media, 2012.
[27] M. Soeken et al. *The EPFL Logic Synthesis Libraries*. 2019. arXiv: 1805.05121v2 [cs.LO].
[28] J. Trommer et al. “Material Prospects of Reconfigurable Transistor (RFETs)–From Silicon to Germanium Nanowires”. In: *MRS* (2014).