Logic Resynthesis of Majority-Based Circuits by Top-Down Decomposition

Siang-Yun Lee
Integrated Systems Lab, EPFL
Lausanne, Switzerland

Heinz Riener
Integrated Systems Lab, EPFL
Lausanne, Switzerland

Giovanni De Micheli
Integrated Systems Lab, EPFL
Lausanne, Switzerland

***Abstract*—Logic resynthesis is the problem of finding a dependency function to re-express a given Boolean function in terms of a given set of divisor functions. In this paper, we study logic resynthesis of majority-based circuits, which is motivated by the increasing interest in majority logic optimization due to the recent development of beyond-CMOS technologies. To meet the need for an efficient majority resynthesis heuristic, we propose a top-down decomposition algorithm, whose complexity is linear to both n and m, where n is the number of divisors and m is the number of majority operations in the dependency function. We evaluate the resynthesis algorithms by using them in a resubstitution run applied on the EPFL benchmark suite. The experimental results show that, comparing to the state-of-the-art enumeration algorithm whose complexity grows exponentially with m, using the proposed decomposition algorithm leads to 1.5% more circuit size reduction by lifting the limitation on m, within comparable runtime.**

***Index Terms*—Logic synthesis, combinational circuit, peephole optimization, majority-inverter graph, Boolean resubstitution**

I. Introduction

Majority-based circuits are gaining interests recently due to their applications in emerging technologies such as the adiabatic quantum flux parametron (AQFP) [1], spin-wave devices [2], and quantum-dot cellular automata (QCA) [3]. As the computation of these technologies are inherently based on the majority function [4], the *majority-inverter graph* (MIG) [5] is a natural choice for the technology-independent representation during logic synthesis and optimization for them [6]. Existing optimization techniques targeting MIGs include algebraic transformation [5], safe bit-error insertion [5], cut rewriting with functional hashing [7], look-up table (LUT) mapping [8], detecting functional equivalences with don't-care consideration by node merging [9], and Boolean resubstitution [10]. Despite these, the performance of size optimization for MIGs is still lagging behind the heavily-researched counterpart in the classical two-input AND/OR-based logic synthesis.

In this work, we focus on *logic resynthesis* for MIGs, which arises from peephole logic optimization algorithms, which are usually Boolean methods capable of utilizing don't-care information, such as Boolean resubstitution [11], [12]. These algorithms focus on a small portion of the circuit at a time and try to locally optimize the subcircuit by resynthesizing it. To this end, *logic resynthesis*, or simply *resynthesis*, is the problem of finding a Boolean function, called the *dependency function*, which takes some given functions, called the *divisors*, as inputs and produces a desired function, called the *target*, at its output. In addition, as we target size optimization for MIGs, dependency functions corresponding to smaller MIGs, called the *dependency circuit*, are particularly favored.

Pioneering research on the theory of majority functions [4] and majority synthesis [13] date back to the 60s. In [13], a majority synthesis algorithm was proposed, which builds up the circuit bottom-up from primary inputs to simple gates and finally realizes the target. However, this algorithm is based on high-complexity bit manipulations which cannot be easily parallelized. The original paper did not include any experimental result, and our complexity analysis and experimental assessment show that its efficiency is not promising in practice. An alternative approach to logic resynthesis is based on enumeration. This method was proposed in [11] for AIGs and adapted for MIGs in [10]. By fixing the structure of the dependency circuit, enumerating all the possible combinations of divisors connected to its inputs, and comparing the output function to the target, the enumeration-based algorithm shows good quality in finding optimal dependency circuits composed of a few gates. However, its complexity is exponential in the number of divisors, with the size of the dependency circuit being in the exponent. This algorithm is thus limited and cannot be further scaled up.

To address the drawbacks of the existing algorithms, in this paper, we propose an efficient top-down decomposition algorithm for majority resynthesis. Decomposition is a common approach in logic synthesis [14], [15], which tries to decompose the given target into simpler or easier-to-synthesize forms. The proposed algorithm is based on computing the "care" bits for each fanin of a node where we need it to be the same as the target and counting the number of care bits each candidate covers.

The experimental results show that, without limitation on the size of the dependency circuit, using the proposed decomposition in resubstitution gives 9.99% circuit size reduction on average, while the state-of-the-art enumeration-based resubstitution [10] can achieve at most 8.45% average reduction within comparable runtime.

This research was supported in part by the EPFL Open Science Fund and by the SNF grant "Supercool: Design methods and tools for superconducting electronics," 200021_1920981.

978-1-6654-3595-6/21/$31.00 ©2021 European Union

II. Preliminaries

In this paper, all functions are, unless otherwise specified, single-output *Boolean functions*, which are functions defined over the Boolean space $\mathbb{B} = \{0, 1\}$.

A. Logic Resynthesis

Logic resynthesis (or simply *resynthesis*) is the problem of re-expressing a function in terms of other functions. More precisely, the *resynthesis problem* is stated as follows: Given a target function $f : \mathbb{B}^k \to \mathbb{B}$ over k Boolean variables $\vec{x} = x_1, \ldots, x_k$ and a collection $G = \{g_1, \ldots, g_n\}$ of n existing functions $g_i : \mathbb{B}^k \to \mathbb{B}$ over the same variables, find a *dependency function* $h : \mathbb{B}^n \to \mathbb{B}$ satisfying

$$f(\vec{x}) = h(g_1(\vec{x}), \ldots, g_n(\vec{x})). \tag{1}$$

for all $\vec{x} \in \mathbb{B}^k$. In this formulation, variables $x_1, \ldots, x_k$ are not the direct inputs of the function h, but any subset of them can be embedded by defining $g_1(\vec{x}) = x_1$, $g_2(\vec{x}) = x_2$, and so forth. Also, the Boolean expression of h does not necessarily depend on all of its n inputs. In the remaining of this paper, the function f is called the *target* and the functions $g_i \in G$ are referred to as *divisors*.

With this formulation, the resynthesis problem can be seen as a generalization of the classical *logic synthesis* problem, where the inputs of h are typically the same Boolean variables $x_1, \ldots, x_k$. Logic resynthesis is different from *logic decomposition* [14], [15] or *functional decomposition* [16], [17], where the problem also involves identifying the divisors g_i.

B. Majority-Inverter Graphs

Logic networks, or simply *networks*, are technology-independent representations of digital circuits used as the data structure during logic optimization. Networks are directed acyclic graphs, where nodes represent logic gates and edges represent interconnecting wires. Incoming edges of a node are called *fanins*, whereas outgoing edges are called *fanouts*. For convenience, the fanins of a node are said to be *siblings* of each other. In this paper, we focus on *majority-inverter graphs* (MIGs) [5], which are networks where each node represents a three-input majority (MAJ) gate and edges can be complemented to represent inverters. Each MAJ gate in the network computes the majority function M of its fanins [4], i.e.,

$$M(x_1, x_2, x_3) = (x_1 \wedge x_2) \vee (x_2 \wedge x_3) \vee (x_1 \wedge x_3). \tag{2}$$

The size of a network is determined by its number of nodes, and inverters are not counted towards the network size.

C. Truth Tables and Bit Operations

We use truth tables to describe single-output Boolean functions and to manipulate them. The *truth table* $T[f] = u_1 \cdots u_l$ of a Boolean function $f : \mathbb{B}^k \to \mathbb{B}$ over k Boolean variables is a string of $l = 2^k$ bits $u_i \in \mathbb{B}$. The bit u_i at the i-th position is equal to the output of $f(\vec{a})$ for the input assignment $\vec{a} = a_1, \ldots, a_k$, where

$$2^{k-1} \cdot a_k + \ldots + 2^0 \cdot a_1 = i. \tag{3}$$

Since the *length* l of a truth table $T[f]$ grows exponentially with the number k of Boolean variable in f, i.e., $l = 2^k$, truth tables are typically only used to represent functions if k is small ($k < 16$).

We use $T[f]_i$ to denote the i-th bit in the truth table of a function f, and

$$\#1(f) = \sum_{i=1}^{l} T[f]_i. \tag{4}$$

to denote the number of 1-bits of the truth table of f.

III. The Top-Down Decomposition Algorithm

In this section, we propose a heuristic algorithm for majority resynthesis using a top-down decomposition. The algorithm receives a target and a set of divisors as inputs and returns a dependency circuit composed of MAJ gates. The term *top-down* indicates how the dependency circuit is constructed: The algorithm gradually "refines" the dependency circuit by adding more nodes to the bottom.

Before we describe the algorithm in detail in the following sections, we give a brief overview of the overall design decisions:

1) *Tree-like circuit structure*: We restrict the dependency circuit to be tree-like, i.e., we forbid logic sharing of the constructed nodes such that each majority gate is used only once as fanin.
2) *Input inversion*: Due to the self-duality property of the majority function [4], inverters can always be pushed to the primary inputs. Hence, we can limit our search to circuit structures without internal inverters by considering that only divisors at the leaves can be complemented. Moreover, the consideration of input inversions is made implicit by supplementing the divisor set G with complemented divisors.
3) *Usage of truth table*: In our algorithm, all functions are represented by truth tables of the same length, i.e., the target and the divisors are truth tables and each node in the constructed dependency circuit has a truth table that represents its function.
4) *Normalization*: As in [13], the divisors are normalized by computing their XNOR ($\leftrightarrow$) with the target. By doing so, the logic of the algorithm is simplified—comparing the output function of the dependency circuit with the target simplifies to testing if the output function is equivalent to the constant 1 function.

Given the target f and the set of divisors $G = \{g_1, \ldots, g_n\}$, the set of normalized divisors to be chosen from as inputs to the dependency circuit is denoted by

$$D = \{d_{2i-1} = g_i \leftrightarrow f,\ d_{2i} = \bar{g}_i \leftrightarrow f \mid 1 \leq i \leq n\}. \tag{5}$$

A. The Care Function

Consider a node with function $f_n = M(f_1, f_2, f_3)$ and a certain bit position p in its truth table. In order to have $T[f_n]_p = 1$, we must have

$$T[f_i]_p = T[f_j]_p = 1, \text{ where } i, j \in \{1, 2, 3\} \text{ and } i \neq j.$$

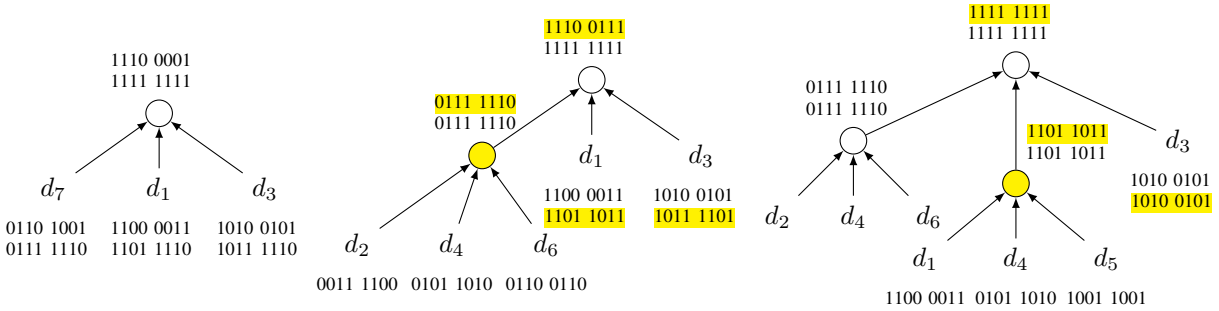

Fig. 1. Top-down decomposition of the target $f(\vec{x}) = x_1 \oplus x_2 \oplus x_3$ using divisors $G = \{g_1(\vec{x}) = x_1, g_2(\vec{x}) = x_2, g_3(\vec{x}) = x_3, g_4(\vec{x}) = 0\}$. The (upper) truth tables represent either the divisor functions normalized with Equation (5) or the output functions of the nodes. The care functions, if relevant, are presented below as the second truth table. The yellow-shaded parts are those being updated after the expansion. The final solution is $h(g_1, g_2, g_3, g_4) = M(M(\bar{g}_1, \bar{g}_2, \bar{g}_3), M(g_1, \bar{g}_2, g_3), g_2)$.

A bit position p is said to be a *care bit* for a fanin edge if $T[f_{s1}]_p = 0$ or $T[f_{s2}]_p = 0$ is known, where f_{s1} and f_{s2} are the sibling functions of the concerned fanin. Extending to all positions, we define the *care function* of a fanin edge as

$$c = (\bar{f}_{s1} \vee \bar{f}_{s2}) \wedge c_n, \tag{6}$$

where f_{s1} and f_{s2} are the sibling functions and c_n is the care function of the node the edge points to. The care function *of a node* is the care function computed for its unique fanout edge, as a fanin of another node. For the topmost node, the care function is simply constant 1 because all bits are under concern. A care bit of a fanin edge is said to be *covered* if the fanin is coming from a divisor or a node whose truth table indeed provides a 1 at this bit. To achieve this with a node, at least two fanins of the node have to *cover* the bit by providing a 1 in their truth tables.

B. Choosing Divisors

We start with a heuristic to choose three divisors f_1, f_2, f_3 from D as fanins of a node with function $f_n = M(f_1, f_2, f_3)$ and care function c_n, aiming at maximizing $\#1(f_n \wedge c_n)$.

$$f_1 = \operatorname*{argmax}_{d \in D}(\#1(d \wedge c_n))$$
$$f_2 = \operatorname*{argmax}_{d \in D_2}(\#1(f_1 \wedge d \wedge c_n) + 2 \cdot \#1(\bar{f}_1 \wedge d \wedge c_n))$$
$$f_3 = \operatorname*{argmax}_{d \in D_3}(\#1((f_1 \oplus f_2) \wedge d \wedge c_n) + 2 \cdot \#1((\bar{f}_1 \wedge \bar{f}_2) \wedge d \wedge c_n))$$
$$\text{where } D_2 = D \backslash \{f_1, \bar{f}_1\}, D_3 = D_2 \backslash \{f_2, \bar{f}_2\}$$

The first divisor is chosen to cover the most care bits for the first time. When choosing the second divisor, the care bits covered by the first divisor still have to be covered again ($f_1 \wedge d \wedge c_n$). But more importantly and thus the doubled weight, the care bits that are not covered by the first divisor are more eager to be covered ($\bar{f}_1 \wedge d \wedge c_n$). For the last divisor, the care bits that are already covered twice can be ignored; the care bits covered only once ($(f_1 \oplus f_2) \wedge d \wedge c_n$) seek to be covered again; the care bits that are never covered before ($(\bar{f}_1 \wedge \bar{f}_2) \wedge d \wedge c_n$) appear to be more difficult to cover than the other bits and they are thus doubly weighed. In the last case, it may seem non-intuitive to cover these bits with the last divisor because covering them only once is not enough. However, the first two divisors may be replaced by new nodes later on in the algorithm. Hence, it is useful to have covered them at least once.

C. Expansion

When all care bits of the three fanins of the topmost node are covered, the constant 1 function is successfully derived at its output and the algorithm terminates. After constructing the first node with three divisors, we choose one of the fanins with uncovered care bits, if any, and try to cover more care bits by replacing the divisor with a new node. This process is called an *expansion*.

To *expand* a fanin, the original divisor is temporarily taken away. Then, three divisors are chosen as the fanins of the new node with the same heuristic as in Section III-B. After an expansion, the function provided to the expanded fanin is different, and the care functions of its siblings are updated accordingly. Until constant 1 is derived at the output of the topmost node by covering all the care bits, the algorithm proceeds by choosing another position to expand. An *expansion position* is a fanin of any node which is connected to a divisor and whose care function is not fully covered. Heuristically, we choose the position with the least uncovered care bits to be expanded first because it is closest to be fully covered.

It is possible that the majority output of the three chosen divisors does not cover more care bits than the original divisor. Hence, the new node is only constructed and used to replace the original divisor if the number of covered care bits increases. When an expansion position is tried but the coverage of care bits does not increase, the new node is discarded and the position is marked as visited to avoid trying it again. However, if its care function is updated because of an update in the function of one of its siblings, the visited flag is reset and the expansion position may be tried again. To avoid constructing nodes with the same divisors repeatedly in a chain, when the care function of a node is the same as one of its fanins, the expansion position at this fanin is directly marked as visited without trying to expand it.

Input: target f, divisors $G = \{g_1, \ldots, g_n\}$
Output: dependency circuit H

```
1  D ← normalize(f, G)
2  n_0 ← choose_divisors(1, D)
3  H ← {n_0}
4  while n_0.output ≠ 1 do
5      (n_p, i) ← choose_expansion_position(H)
6      n ← choose_divisors(n_p.fanin(i).care, D)
7      if accept_expansion(n_p, i, n) then
8          n_p.fanin(i) ← n
9      else
10         mark_visited(n_p, i)
11 return H
```

Algorithm 1: Maj. resynthesis by top-down decomposition.

D. Quality and Efficiency Enhancements

1) Computation Cache: The heuristic presented in Section III-B is used many times during the execution of the algorithm, and its result depends solely on the care function c_n. To avoid repeating the same computation, a computed table can be used to cache the results, implemented with a hash table mapping from a care function to three divisors.

2) Exhaustive Trials for the Topmost Node: In general, the first few choices of a heuristic algorithm usually have a greater impact on the quality of the final result. For better quality, exhaustive trials are applied for the topmost node as it largely determines the overall decomposition. These include: (1) when choosing the three divisors for the first node, all the divisors which score the highest; and (2) all of the three fanins of the topmost node, which to be expanded first. All possibilities are tried independently and the best result is returned as the final solution.

E. Summary

Algorithm 1 illustrates the decomposition algorithm. Procedure *normalize* (line 1) derives the set D of normalized divisors by Equation (5). Procedure *choose_divisors* (lines 2 and 6) builds a new node by choosing three divisors from D with the scoring functions described in Section III-B or looking up in the computation cache as described in Section III-D1. The first argument to *choose_divisors* is the care function, which is constant 1 for the topmost node and is computed by Equation (6) otherwise. Procedure *choose_expansion_position* (line 5) returns a pair of a parent node n_p and a fanin index $i \in \{1, 2, 3\}$ representing an expansion position chosen as described in Section III-C. The condition in line 7 decides if the expansion is accepted by whether the coverage of care bits increases with the new node n. If so, fanin i of the parent node n_p is replaced by n (line 8); otherwise, the expansion position is marked as visited (line 10). The expansion attempt is repeated until constant 1 is derived at the output of the topmost node n_0. Figure 1 shows an execution example of the algorithm with two expansions.

IV. Comparisons to Existing Algorithms

In this section, we briefly introduce the two existing algorithms for majority resynthesis mentioned in the introduction, Akers' algorithm and enumeration, and provide theoretical analysis and comparisons with our proposed algorithm.

TABLE I
Comparisons of three resynthesis algorithms.

	Akers' algorithm	Enumeration	Decomposition
Soundness	yes	yes	yes
Completeness	only for $m \leq 1$	yes/no	no
Optimality	only for $m \leq 1$	yes	no
Complexity	$O(n^2ml^2)$	$O(n^{2m+1}l)$	$O(nml)$

Akers' majority synthesis algorithm [13] adopts a bottom-up approach that builds new nodes from the constructed ones. This algorithm is based on iterative row and column manipulations on a binary table, where each column corresponds to a divisor or a constructed node and each row corresponds to a bit position in their truth tables. Akers' algorithm is an heuristic and is not guaranteed to terminate if no resource constraint is given. In contrast, the recent alternative, proposed in the context of MIG resubstitution [10], tries to find the optimal solution if it is small enough. This algorithm enumerates dependency circuits of at most one or two nodes with all possible combinations of divisors connected to their inputs and compares their output function to the target. Some filters can be applied to enhance its efficiency, but the algorithm is, in its nature, difficult to be extended for finding larger solutions in terms of reasonable runtime as well as algorithm implementation.

In the following analysis, n is the number of divisors, m is the number of nodes in the dependency circuit, $l = 2^k$ is the length of the truth tables and k is the number of variables of the target and divisors. Table I compares the characteristics and complexities of the three algorithms. We omit the details on complexity analysis due to the limited space. All of the three algorithms are *sound*, meaning that the dependency circuits they produce, if any, indeed realize the given target correctly. However, *completeness* is not always achievable—none of the algorithms guarantees to find solutions to all the problems for which solutions exist, unless $m \leq 1$. For enumeration, completeness depends on the implementation choice on the tradeoff between efficiency and quality. When exhaustive enumeration is too time-consuming, over-filtering to reduce the search space may be used, sacrificing completeness. Lastly, the *optimality* discussed here is with respect to minimal m. Enumeration is optimal when it finds a solution, whereas the two heuristics do not always guarantee optimality.

V. Experimental Results

The three majority resynthesis algorithms are implemented in C++-17 as part of the EPFL logic synthesis library *mockturtle*[1] [18]. We evaluate the proposed algorithm and compare to existing algorithms by using them to solve the resynthesis problem in a Boolean resubstitution framework similar to the one proposed in [11]. Resubstitution is applied on MIGs derived from the EPFL combinational benchmark suite [19] using windows with at most 8 inputs. The experiments are performed

[1] Available: github.com/lsils/mockturtle

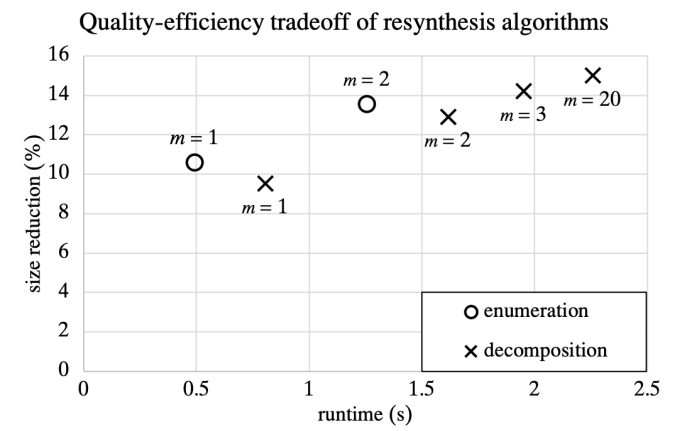

Fig. 2. Average size reduction and runtime of resubstitution using enumeration and the proposed top-down decomposition as the internal resynthesis engine and with different m values.

on a Linux machine with Xeon 2.5 GHz CPU and 256 GB RAM.

In the following tables, the gain is measured as the difference of the network sizes before and after one run of resubstitution. Column (%) shows the percentage of network size reduction with respect to its original size. The overall runtime in seconds is shown in column (t). The parameter m is the user-specified upper bound on the number of nodes in the dependency circuit. When m is set to a larger value, the size of the dependency circuit is more often bounded by the size of the subnetwork to be replaced, which is the size of the *maximum fanout-free cone* (MFFC) [20] in resubstitution.

A. Evaluation of the Existing Algorithms

First, we provide experimental evaluation of our re-implementation of the existing algorithms analyzed in Section IV. The resubstitution results using them as the internal resynthesis engine are shown in Table II. As can be seen from the table, Akers' algorithm is inefficient due to its high complexity. It is thus excluded for comparison in the following sections. With $m = 1$, the enumeration-based resubstitution achieves an average size reduction of 10.58% in 0.49 seconds. The runtime is much better than in the original paper [10] because of implementation improvements.

B. Quality-Efficiency Tradeoff

In this section, we compare the proposed top-down decomposition algorithm against enumeration and examine their quality-efficiency tradeoff curves. Figure 2 shows the size reduction and runtime, averaged over all benchmarks, of resubstitution runs using the two algorithms as the internal resynthesis engine and with different settings of the parameter m. The results using enumeration are plotted with circles, whereas those using decomposition are plotted with crosses.

There is a tradeoff between quality and efficiency for both algorithms. Although the trending curve of decomposition lies below that of enumeration, decomposition is easier to be extended further to achieve better quality when higher runtime is affordable. This is for two reasons: (1) the implementation of enumeration is hard to be generalized for any m because different circuit structures are separated coded; and (2) each increment of m increases the complexity of the algorithm in the exponent of n (number of divisors) by 2 (Table I).

C. Resubstitution Quality on Pre-optimized Benchmarks

Resubstitution is often used as a fine-tuning optimization step after applying some larger-scale optimization algorithms such as cut rewriting [7] or LUT-mapping [8]. To capture the impact of different resynthesis algorithms in resubstitution in a more practical scenario, in this section, we compare the quality of resubstitution applied on benchmarks that are pre-optimized with three iterations of cut rewriting.

Table III shows the resubstitution results with the proposed algorithm compared to enumeration. Since the benchmarks are already optimized, there is less space left for simple resubstitution. This highlights the advantages of the heuristic decomposition algorithm. With $m = 2$ for both algorithms, decomposition achieves higher average size reduction (9.46% as opposed to 8.45%) in lower average runtime (0.59 seconds as opposed to 0.67 seconds). Extending to $m = 3$, the proposed algorithm obtains 9.99% average size reduction in a still-comparable runtime of 0.81 seconds.

VI. Conclusions

Motivated by its need in peephole optimization, we study logic resynthesis for majority-based circuits in this paper. Our results show that Akers' algorithm is inefficient and that enumeration is not scalable with respect to m. Hence, we propose a top-down decomposition algorithm whose complexity is linear to the number of divisors (n), the length of truth tables (l), and size of the dependency circuit (m). Using the proposed resynthesis algorithm in a resubstitution framework, our experimental results show that it provides good quality and efficiency comparable to enumeration with small m values. Moreover, it is more flexible to be extended for larger m values to achieve better optimization quality. On a set of pre-optimized benchmarks, the proposed algorithm leads to 1.5% more circuit size reduction (from 8.45% to 9.99%) in 1.3x runtime (from 0.67 to 0.81 seconds), comparing to the state-of-the-art enumeration-based resubstitution. Our future work includes improving the decomposition algorithm to consider logic sharing and leveraging don't-care information in resynthesis and resubstitution.

References

[1] R. Cai, O. Chen, A. Ren, N. Liu, C. Ding, N. Yoshikawa, and Y. Wang, "A majority logic synthesis framework for adiabatic quantum-flux-parametron superconducting circuits," in *Proceedings of GLSVLSI*, 2019, pp. 189–194.

[2] A. Khitun and K. L. Wang, "Non-volatile magnonic logic circuits engineering," *Journal of Applied Physics*, vol. 110, no. 3, p. 034306, 2011.

[3] C. S. Lent, P. D. Tougaw, W. Porod, and G. H. Bernstein, "Quantum cellular automata," *Nanotechnology*, vol. 4, no. 1, p. 49, 1993.

[4] S. Muroga, I. Toda, and S. Takasu, "Theory of majority decision elements," *Journal of the Franklin Institute*, vol. 271, no. 5, pp. 376–418, 1961.

TABLE II
RESUBSTITUTION RESULTS USING AKERS' ALGORITHM AND ENUMERATION.

		Akers' algorithm			Enumeration					
		$m=1$			$m=1$			$m=2$		
benchmark	size	gain	(%)	t (s)	gain	(%)	t (s)	gain	(%)	t (s)
adder	1020	127	12.45	3.8	127	12.45	0.01	132	12.94	0.02
bar	3336	380	11.39	12.13	392	11.75	0.02	448	13.43	0.10
div	57247	4796	8.38	> 60	5333	9.32	0.95	12256	21.41	2.64
hyp	214335	15012	7.00	> 60	15519	7.24	6.80	24004	11.20	11.64
log2	32060	39	0.12	> 60	62	0.19	0.24	1775	5.54	0.90
max	2865	25	0.87	6.77	25	0.87	0.01	26	0.91	0.03
multiplier	27062	218	0.81	> 60	266	0.98	0.23	1633	6.03	1.11
sin	5416	55	1.02	33.3	99	1.83	0.08	281	5.19	0.89
sqrt	24618	3823	15.53	> 60	4208	17.09	0.34	4216	17.13	0.41
square	18484	216	1.17	> 60	424	2.29	0.16	1438	7.78	0.47
arbiter	11839	128	1.08	13.08	128	1.08	0.06	128	1.08	0.12
cavlc	693	14	2.02	9.58	78	11.26	0.09	107	15.44	4.72
ctrl	174	14	8.05	4.04	88	50.57	0.07	90	51.72	0.21
dec	304	0	0.00	7.97	0	0.00	0.00	0	0.00	0.00
i2c	1342	62	4.62	3.8	97	7.23	0.02	116	8.64	0.05
int2float	260	15	5.77	1.34	38	14.62	0.01	43	16.54	0.13
mem_ctrl	46836	2298	4.91	> 60	3389	7.24	0.58	3744	7.99	1.21
priority	978	240	24.54	4.48	310	31.70	0.01	324	33.13	0.02
router	257	0	0.00	1.13	0	0.00	0.00	11	4.28	0.00
voter	13758	2427	17.64	> 60	3296	23.96	0.17	4219	30.67	0.47
average		1494.45	6.37		1693.95	10.58	0.49	2749.55	13.55	1.26

TABLE III
RESUBSTITUTION RESULTS ON OPTIMIZED BENCHMARKS.

		Enumeration						Decomposition								
		$m=1$			$m=2$			$m=1$			$m=2$			$m=3$		
benchmark	size	gain	(%)	t (s)	gain	(%)	t (s)	gain	(%)	t (s)	gain	(%)	t (s)	gain	(%)	t (s)
adder	512	0	0.00	0.00	1	0.20	0.01	0	0.00	0.01	128	25.00	0.01	128	25.00	0.01
bar	3079	134	4.35	0.02	265	8.61	0.07	134	4.35	0.05	325	10.56	0.13	325	10.56	0.20
div	35993	227	0.63	0.19	237	0.66	0.37	241	0.67	0.47	283	0.79	0.88	341	0.95	0.92
hyp	166673	4393	2.64	1.76	12142	7.28	3.50	4317	2.59	3.12	8150	4.89	4.30	11856	7.11	6.08
log2	30404	22	0.07	0.24	1229	4.04	0.80	22	0.07	0.56	1101	3.62	1.50	1102	3.62	2.20
max	2321	20	0.86	0.01	20	0.86	0.03	20	0.86	0.04	20	0.86	0.12	23	0.99	0.14
multiplier	24532	57	0.23	0.21	992	4.04	0.99	57	0.23	0.45	802	3.27	0.84	802	3.27	1.22
sin	4958	35	0.71	0.10	108	2.18	0.83	36	0.73	0.13	111	2.24	0.29	121	2.44	0.36
sqrt	13449	2099	15.61	0.08	2097	15.59	0.28	1979	14.71	0.19	1979	14.71	0.29	1979	14.71	0.31
square	17255	220	1.27	0.15	1021	5.92	0.41	206	1.19	0.28	1025	5.94	0.47	1172	6.79	0.53
arbiter	11839	128	1.08	0.06	128	1.08	0.12	128	1.08	0.13	128	1.08	0.35	128	1.08	0.73
cavlc	656	59	8.99	0.10	86	13.11	4.31	54	8.23	0.05	84	12.80	0.11	100	15.24	0.14
ctrl	127	46	36.22	0.03	46	36.22	0.33	42	33.07	0.01	45	35.43	0.02	45	35.43	0.01
dec	304	0	0.00	0.01	0	0.00	0.01	0	0.00	0.02	0	0.00	0.02	0	0.00	0.02
i2c	1260	88	6.98	0.02	95	7.54	0.05	104	8.25	0.03	128	10.16	0.06	144	11.43	0.06
int2float	217	12	5.53	0.01	14	6.45	0.08	9	4.15	0.01	18	8.29	0.03	20	9.22	0.03
mem_ctrl	43045	1842	4.28	0.42	2803	6.51	1.02	1481	3.44	0.78	2288	5.32	1.82	2968	6.90	2.59
priority	896	199	22.21	0.01	207	23.10	0.06	92	10.27	0.06	101	11.27	0.32	106	11.83	0.32
router	245	1	0.41	0.00	1	0.41	0.01	0	0.00	0.01	1	0.41	0.02	1	0.41	0.04
voter	7292	1439	19.73	0.05	1836	25.18	0.14	1541	21.13	0.11	2381	32.65	0.20	2386	32.72	0.25
average		551.05	6.59	0.17	1166.4	8.45	0.67	523.15	5.75	0.33	954.9	9.46	0.59	1187.35	9.99	0.81

[5] L. Amaru, P.-E. Gaillardon, and G. De Micheli, "Majority-inverter graph: A new paradigm for logic optimization," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 35, no. 5, pp. 806–819, 2015.

[6] E. Testa, S.-Y. Lee, H. Riener, and G. De Micheli, "Algebraic and Boolean optimization methods for AQFP superconducting circuits," in *Proceedings of ASP-DAC*, 2021, pp. 779–785.

[7] M. Soeken, L. G. Amarù, P.-E. Gaillardon, and G. De Micheli, "Optimizing majority-inverter graphs with functional hashing," in *Proceedings of DATE*. IEEE, 2016, pp. 1030–1035.

[8] W. J. Haaswijk, M. Soeken, L. Amaru, P.-E. Gaillardon, and G. De Micheli, "LUT mapping and optimization for majority-inverter graphs," in *Proceedings of IWLS*, 2016.

[9] C.-C. Chung, Y.-C. Chen, C.-Y. Wang, and C.-C. Wu, "Majority logic circuits optimisation by node merging," in *Proceedings of ASP-DAC*. IEEE, 2017, pp. 714–719.

[10] H. Riener, E. Testa, L. Amaru, M. Soeken, and G. De Micheli, "Size optimization of MIGs with an application to QCA and STMG technologies," in *Proceedings of Symposium on Nanoscale Architectures*, 2018, pp. 157–162.

[11] A. Mishchenko and R. K. Brayton, "Scalable logic synthesis using a simple circuit structure," in *Proceedings of IWLS*, 2006.

[12] A. Mishchenko, R. K. Brayton, J.-H. R. Jiang, and S. Jang, "Scalable don't-care-based logic optimization and resynthesis," *ACM Transactions on Reconfigurable Technology and Systems*, vol. 4, no. 4, pp. 1–23, 2011.

[13] S. B. Akers, "Synthesis of combinational logic using three-input majority gates," in *Proceedings of Symposium on Switching Circuit Theory and Logical Design*. IEEE, 1962, pp. 149–158.

[14] V. Bertacco and M. Damiani, "The disjunctive decomposition of logic functions," in *Proceedings of ICCAD*, vol. 97, 1997, pp. 78–82.

[15] A. Mishchenko, B. Steinbach, and M. Perkowski, "An algorithm for bi-decomposition of logic functions," in *Proceedings of DAC*, 2001, pp. 103–108.

[16] Z. Chu, M. Soeken, Y. Xia, and G. De Micheli, "Functional decomposition using majority," in *Proceedings of ASP-DAC*. IEEE, 2018, pp. 676–681.

[17] Y.-T. Lai, K.-R. Pan, and M. Pedram, "OBDD-based function decomposition: Algorithms and implementation," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 15, no. 8, pp. 977–990, 1996.

[18] M. Soeken, H. Riener, W. Haaswijk, E. Testa, B. Schmitt, G. Meuli, F. Mozafari, and G. De Micheli, "The EPFL logic synthesis libraries," *arXiv preprint arXiv:1805.05121v2*, 2019.

[19] L. Amarú, P.-E. Gaillardon, and G. De Micheli, "The EPFL combinational benchmark suite," in *Proceedings of IWLS*, no. CONF, 2015.

[20] J. Cong and Y. Ding, "On area/depth trade-off in LUT-based FPGA technology mapping," *IEEE Transactions on Very Large Scale Integration Systems*, vol. 2, no. 2, pp. 137–148, 1994.