Exploiting Coding Techniques for Logic Synthesis of Reversible Circuits

Alwin Zulehner Robert Wille
Institute for Integrated Circuits, Johannes Kepler University Linz, Austria
alwin.zulehner@jku.at robert.wille@jku.at

Abstract—Reversible circuits are composed of a set of circuit lines that are passed through a cascade of reversible gates. Since the number of circuit lines is crucial, functional logic synthesis approaches have been proposed which realize circuits where the number of circuit lines is minimal. However, since the function to be realized is often non-reversible, additional variables have to be added to the function in order to establish reversibility – leading to a significant overhead that affects the scalability of the synthesis method and yields rather complex circuits. In this work, we propose to overcome these problems by exploiting coding techniques in the logic synthesis of reversible circuits. To this end, we propose an intermediate encoding of the output patterns that requires fewer additional inputs and outputs. Using this synthesis scheme allows to perform the majority of the synthesis on significantly fewer variables and to exploit several *don't care* values in the code. Experimental evaluations – where we obtain better scalability and circuits with magnitudes fewer costs – confirmed the benefits of the proposed synthesis approach.

I. Introduction

Reversible logic allows not only for performing computation from the inputs towards the outputs, but also for recovering the inputs from the outputs. Originally, this computation paradigm was motivated due to its relation to low-power computing (cf. [10, 4, 5]) and its application in the domain of quantum computing [14]. However, in the recent years further application areas evolved, including the design of on-chip interconnects [23, 26] and encoders [30] in general, as well as verification [2] – which makes reversible logic (and, thus, reversible circuits) a highly investigated research area.

Reversible circuits have a complementarily different structure compared to conventional circuitry, since they are – due to the non-existence of a direct realization of fan-out and feedback – composed of a set of circuit lines that are passed through a cascade of reversible gates. Consequently, new design techniques are required for synthesizing reversible circuits.

First accomplishments in design automation for reversible circuit synthesis were published more than a decade ago (cf. [13]). Since then, several methods have been proposed aiming for better scalability or for reducing the complexity of the resulting circuits. *Structural approaches* (e.g. [22, 6]), i.e. approaches that map building blocks of conventional data-structures for function representation such as BDDs, ESoPs, etc. to their reversible counterpart, turned out to be the most scalable and yield rather cheap circuits (with respect to commonly applied cost metrics). However, they generate circuits where the number of circuit lines is magnitudes above the minimum (as e.g. evaluated in [25]). The number of circuit lines is crucial, since each circuit line has to be represented physically in the underlying technology. Therefore, *functional approaches* that allow for synthesis of circuits with the minimum number of circuit lines have been proposed. These methods consider the whole function to be synthesized during synthesis, which leads to a less scalable synthesis and rather complex circuits [7, 13, 8]. But also here, significant progress in improving scalability and in reducing the complexity of the resulting circuits has been made recently (cf. [19, 18, 16]).

However, it turns out that the side-effects of the so-called *embedding* process more and more becomes the bottleneck for functional synthesis of reversible circuits. Embedding is needed in order to transform a possibly non-reversible function to be synthesized into a reversible one [11, 25, 20, 28] – a prerequisite for functional synthesis. In this process, additional inputs and outputs are added to the function to make all occurrences of an output pattern distinguishable. This number of additional inputs and outputs is obviously dominated by the most frequent output pattern (this is covered in more detail later in Section III). Even though embedding has been proven to be a coNP-hard problem [20], rather scalable approaches have been proposed recently [20, 28]. However, since the synthesis becomes exponentially harder with each more variable, adding further inputs and outputs as a result of the embedding process poses a significant threat to the scalability of logic synthesis for reversible circuits. Besides that, also the complexity of the resulting circuit increases since functions with more variables usually require more costly gates to become realized.

In this work, we aim for improving the scalability of functional synthesis and for reducing the complexity of the resulting circuit – while guaranteeing minimality with respect to the number of circuit lines. The main idea is motivated by the fact that many of the additional inputs and outputs added during embedding are only required for few output patterns. Representing these output patterns with a smaller number of output bits saves a significant amount of extra logic. In order to accomplish that coding techniques are employed which determines an intermediate, reversibly embedded, function with significantly fewer additional outputs. While this significantly simplifies the synthesis of the function, a corresponding decoder is additionally required. This, however, can be realized efficiently and with a small number of gates in most of the cases.

Experimental results demonstrate the benefits of the proposed idea. The resulting synthesis approach is much more scalable than the current state of the art. Besides that, we obtain circuits with significant improvements in terms of quantum cost (a commonly applied cost metric for reversible circuits). In fact, reductions of several orders of magnitude have been observed in many cases.

This paper is structured as follows. In Section II, we briefly review reversible functions, their representation, as well as reversible circuits. In Section III, we analyze the current design flow for synthesis of reversible circuits and discuss the open potential that leads to the general idea of this work. Details on an efficient implementation of the proposed scheme are presented in Section IV, whereas Section V summarizes the obtained results. Section VI concludes the paper.

II. Background

In this section, we briefly recap reversible functions, their representation, as well as reversible circuits.

Definition 1 *A Boolean function $f : \mathbb{B}^n \to \mathbb{B}^m$ is reversible, iff $n = m$ and f is bijective.*

Example 1 *Consider the Boolean function represented by means of the truth table shown in Table Ia. This function is reversible, since the number of inputs is equal to the number of outputs and there exists a one-to-one mapping from inputs to outputs, i.e. each pattern occurs once at the input and once at the output.*

Since reversible functions describe a permutation of the input patterns, they are also commonly described by permutation matrices.

Definition 2 *Let $f : \mathbb{B}^n \to \mathbb{B}^n$ be a reversible Boolean function. Then, the permutation matrix M of f is a $2^n \times 2^n$ Boolean matrix where the entries $m_{i,j}$ $(0 \leq i, j < n)$ indicate whether an input (column) is mapped to an output (row), i.e.*

$$m_{i,j} = \begin{cases} 1 & \text{if } f(j) = i, \\ 0 & \text{otherwise.} \end{cases}$$

TABLE I
REVERSIBLE FUNCTION REPRESENTATION

(a) Truth-table (b) Permutation matrix

x_2	x_1	x_2'	x_1'
0	0	1	0
0	1	0	0
1	0	0	1
1	1	1	1

Outputs \ Inputs	00	01	10	11
00	0	**1**	0	0
01	0	0	**1**	0
10	**1**	0	0	0
11	0	0	0	**1**

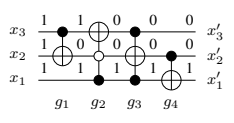

Fig. 1. A reversible circuit

Example 1 (continued) *Consider again the Boolean function shown in Table Ia. The corresponding permutation matrix is shown in Table Ib. Note that there is exactly one 1-entry in each row and in each column. The 1-entry in the second column describes a mapping from input $x_2x_1 = 01$ to output $x_2'x_1' = 00$.*[1]

Reversible functions can be realized by a reversible circuit, which allows to compute not only the outputs from the inputs, but also vice versa. Reversible circuits employ a completely different structure than conventional circuitry. In fact, the variables of the reversible function are represented by means of circuit lines, which are passed through a cascade of reversible gates. These gates may change the value of some circuit lines.

Definition 3 *A reversible circuit $C = (X, G)$ is composed of a set $X = \{x_1, x_2, \dots x_n\}$ of n circuit lines and a sequence $G = g_1, g_2, \dots g_l$ of reversible gates. A commonly used reversible gate is the* Toffoli gate *$g_i = (C_i, t_i)$, which is composed of a set $C_i \subseteq \{x_j^+ | x_j \in X\} \cup \{x_j^- | x_j \in X\}$ of positive and negative control lines and a target line $t_i \in X$ with $\{t_i^-, t_i^+\} \cap C_i = \emptyset$. Furthermore, a circuit line must not occur as positive and negative control line in a gate. The value of the target line t_i is inverted by gate g_i iff all positive (negative) control lines are assigned one (zero). All other lines are passed through the gate unaltered.*

Example 2 *Fig. 1 shows a reversible circuit composed of three circuit lines and four Toffoli gates. Furthermore, the circuit lines are annotated with their respective value when applying input combination $x_3x_2x_1 = 111$. The first gate g_1 inverts the value of target line x_2, because the positive control line x_3^+ is assigned 1. Gate g_2 inverts the value of target line x_3, because the control lines x_2^- and x_1^+ are assigned 0 and 1, respectively. The remaining two gates do not alter the value on any circuit lines, because the control lines are not assigned accordingly.*

The complexity of reversible circuits is usually measured by means of quantum cost. These cost result from a mapping of the reversible gates to a quantum gate library (e.g. *Clifford+T*). In case of the *Clifford+T* library, the quantum cost is then called *T-depth*, which is determined by the number of T-gates that are applied subsequently [3]. Obviously, the quantum cost depends on how the reversible gates are mapped to the respective library. In this paper, we use the latest version of RevKit [17] to determine the quantum cost of reversible gates/circuits. The quantum cost increases with the number of control lines. For example, Toffoli gates with 1, 2, 3, and 4 control lines have a *T-depth* of 0, 3, 12, and 30, respectively.

Example 2 (continued) *The reversible circuit shown in Fig. 1 has a* T-depth *of 6.*

III. CURRENT DESIGN PROCESS AND OPEN POTENTIAL

This section reviews the established process for the functional design of reversible circuits. Based on that, we discuss open potential in this process which motivates the general idea of this work and, eventually, allows for a design process yielding much more compact reversible circuits.

[1] In the following, we will use the term *variable* when referring to the mapping from the i^{th} bit of the input to the i^{th} bit of the output.

TABLE II
EMBEDDING OF A NON-REVERSIBLE FUNCTION

(a) Orig. function

x_3	x_2	x_1	x_3'	x_2'	x_1'
0	0	0	0	1	0
0	0	1	0	1	0
0	1	0	1	0	0
0	1	1	1	0	0
1	0	0	0	1	1
1	0	1	0	1	0
1	1	0	0	1	0
1	1	1	0	0	1

(b) Output patterns

i	p_i	$\mu(p_i)$
1	010	4
2	100	2
3	001	1
4	011	1
5	000	0
6	101	0
7	110	0
8	111	0

(c) Embedded function

a_2	a_1	x_3	x_2	x_1	x_3'	x_2'	x_1'	g_2	g_1
0	0	**0**	**0**	**0**	**0**	**1**	**0**	0	0
0	0	**0**	**0**	**1**	**0**	**1**	**0**	0	1
0	0	**0**	**1**	**0**	**1**	**0**	**0**	0	0
0	0	**0**	**1**	**1**	**1**	**0**	**0**	0	1
0	0	**1**	**0**	**0**	**0**	**1**	**1**	0	0
0	0	**1**	**0**	**1**	**0**	**1**	**0**	1	0
0	0	**1**	**1**	**0**	**0**	**1**	**0**	1	1
0	0	**1**	**1**	**1**	**0**	**0**	**1**	0	0
					⋮			⋮	

A. Current Design Process

Using functional approaches, reversible circuits are usually realized in a two-stage process: embedding followed by the actual functional synthesis. Embedding is required in order to realize non-reversible functions where multiple input assignments are mapped to the same output patterns and, hence, must be made distinguishable to ensure a unique input/output-mapping. To this end, outputs (so-called *garbage outputs*) are added to the function which allow for this distinction. The following definition and example formalizes and illustrates this step, respectively.

Definition 4 *Let $f : \mathbb{B}^n \to \mathbb{B}^m$ be a Boolean function with n inputs and m outputs. Furthermore, let $\mu : \mathbb{B}^m \to \mathbb{N}$ be a function that provides the number of times a respective output pattern $p \in \mathbb{B}^m$ is generated by f. Then, the i^{th} most frequent output pattern ($1 \leq i \leq 2^m$) is denoted by p_i. In order to distinguish all occurrences of an output pattern p_i, at least $k_i = \lceil \log_2 \mu(p_i) \rceil$ garbage outputs are required. Since $k_i \leq k_1$ always holds (i.e. the number of garbage outputs required for an output pattern p_i is always less than or equal to the number of garbage outputs required for the most frequent output pattern p_1), the total number of garbage outputs required for a given function f is $k_1 = \lceil \log_2 \mu(p_1) \rceil$.*

Example 3 *Consider the Boolean function shown in Table IIa. Table IIb lists the output patterns p_i as well as their occurrences determined by μ. Since $p_1 = 010$ is the most frequent output pattern and it occurs $\mu(p_1) = 4$ times, $k_1 = \lceil \log_2 4 \rceil = 2$ garbage outputs are required to make all occurrences of the pattern distinguishable. For pattern $p_2 = 100$, one garbage output is sufficient to distinguish the $\mu(p_2) = 2$ occurrences of this pattern. This garbage outputs is already covered by the two garbage outputs that are demanded by output pattern p_1. All other output patterns do not demand for a garbage output, since they occur at most once.*

Having the additional garbage outputs, the number of inputs of f might have to be adjusted as well. In fact, since the total number of inputs and the total number of outputs must be equal in a reversible function (cf. Def. 1), we need $\max(n, m + k_1)$ inputs. In case of $n < m + k_1$, this requires the addition of so-called *constant inputs*.

Example 3 (continued) *For the considered function f, we have $n = 3$ inputs and $m = 3$ outputs. Since the most frequent output pattern $p_1 = 010$ occurs $\mu(p_i) = 4$ times, we additionally need $k_1 = 2$ garbage outputs. Overall, this yields a reversible function that embeds f (e.g. the one shown in Table IIc), which is composed of a total of $\max(3, 3 + 2) = 5$ variables. By setting the added constant inputs to 0, the non-garbage outputs will realize the desired function f (highlighted bold in Table IIc).*

After the embedding step, a reversible function results which can be synthesized as a reversible circuit. To this end, numerous synthesis methods have been proposed in the past. Since a detailed understanding of the underlying methods of these synthesis approaches is not required to follow the remainder of this work, we refer to the respective related work such as [19, 18, 16] for a more detailed treatment. For our purposes, it is sufficient to understand that the complexity of the synthesis increases with

TABLE III
CODE FOR THE FUNCTION SHOWN IN TABLE IIA

(a) Resulting encoding

i	p_i	$c(p_i)$
1	010	0 - -
2	100	1 0 -
3	001	1 1 0
4	011	1 1 1

(b) Encoded function

x_3	x_2	x_1	x'_3	x'_2	x'_1
0	0	0	0	-	-
0	0	1	0	-	-
0	1	0	1	0	-
0	1	1	1	0	-
1	0	0	1	1	1
1	0	1	0	-	-
1	1	0	0	-	-
1	1	1	1	1	0

the number of variables of the function to be synthesized. This raises the question whether the previously conducted embedding step can be improved in order to ease the synthesis process. The next section discusses open potential in this regard.

B. Open Potential and General Idea of This Work

The established design process as reviewed above employs a rather naive embedding scheme. In fact, it simply adds a total of $k_1 = \lceil \log_2 \mu(p_1) \rceil$ garbage outputs, although, as already discussed by means of Example 3, not all output patterns usually require this amount of garbage outputs. As a consequence, the resulting reversible function (as well as the correspondingly derived reversible circuit) includes a significant amount of extra logic which in many cases is only required for the output pattern p_1. Avoiding this overhead provides significant potential for improving the design process as well as for realizing cheaper circuits.

In this work, we aim to exploit this potential. The main idea is to represent frequently occurring output patterns (which require more garbage outputs) with a smaller number of variables. Vice versa, less frequently occurring patterns (which require less garbage outputs) are represented with a larger number of variables. In other words, coding techniques are utilized in order to encode the desired function with a variable-length code c in which the length of the code word $c(p_i)$ for an output pattern p_i is indirectly proportional to the number $\mu(p_i)$ of times the pattern occurs. An example illustrates the idea.

Example 4 *Consider again the Boolean function shown in Table IIa and its distribution of the output patterns as shown in Table IIb. One possible variable-length code is shown in Table IIIa. There, the most frequent output pattern* $p_1 = 010$ *is encoded by* $c(p_1) = 0$*. Since this pattern requires two garbage outputs, in total* $1 + 2 = 3$ *outputs are required. The garbage outputs are represented by a dash, since they represent* don't care *values (as long as it is ensured that the resulting function is reversible). The second most frequent output pattern* $p_2 = 100$ *is encoded by* $c(p_2) = 10$*. Since this pattern occurs only twice, one garbage output is required – again resulting in* $2 + 1 = 3$ *outputs. The patterns* p_3 *and* p_4 *are encoded by* $c(p_3) = 110$ *and* $c(p_4) = 111$*, respectively. Here, no garbage outputs are required. The remaining patterns (*p_5 *to* p_8*) do not have to be encoded, since they never occur. Overall, this yields an (encoded) reversible function which embeds* f *as shown in Table IIIb and is composed of a total of 3 inputs/outputs only (compared to the original embedding shown in Table IIc which is composed of 5 inputs/outputs).*

Following this idea significantly reduces the number of inputs/outputs and, hence, the number of variables to consider during synthesis. Since the complexity of the synthesis increases exponentially with the number of variables in the function to be synthesized, already this allows for substantial improvements for the synthesis step. Moreover, a nice side-effect occurs by encoding the function to be synthesized. Since the values of the garbage outputs are basically *don't care* (except the restriction that a reversible function has to be realized), there is a significant degree of freedom that can be exploited in order to reduce the complexity of the resulting circuits. This has also been recently exploited in the concept of one-pass synthesis of reversible logic which combines the process of embedding and synthesis [29]. On the downside, a proper variable-length code has to be determined and, after the (encoded) function has been realized as reversible circuit, a corresponding decoding circuit is additionally needed. After applying the decoder, a circuit with the minimum number $\max(n, m + k_1)$ of circuit lines results again. However, the major part of the synthesis task is performed on the encoded function with a significant smaller number of variables. How all these steps can be accomplished in an efficient fashion, while still exploiting the open potential and generating much cheaper circuits, is described in detail in the next section.

IV. IMPLEMENTATION

In order to realize a given function $f : \mathbb{B}^n \rightarrow \mathbb{B}^m$ following the general idea proposed above, three steps have to be conducted: (1) a variable-length code has to be determined, (2) the resulting (encoded) function has to be synthesized as reversible circuit, and (3) a circuit has to be generated which decodes the output of the derived circuit back to the originally intended values. This section provides the implementation details for all these steps.

A. Determining the Code

As discussed above, we aim for a code that allows for splitting the available outputs into a code word and garbage outputs. Doing this splitting individually for each output pattern allows to use short code words (i.e. few primary outputs) for frequently occurring patterns which require a large number of garbage outputs. Vice versa, less frequently occurring output patterns can be encoded with larger code words, since they do not require many garbage outputs. By this, the need for many garbage outputs is compensated by smaller code words. By definition, a Huffman code [9] can guarantee this.

The code is computed by generating the so-called Huffman tree: For each output pattern p_i that occurs at least once in the considered function f (i.e. for each p_i with $\mu(p_i) > 0$), we add a leaf vertex v_i to the Huffman tree. Each of these leaf vertices v_i is associated with a *weight* $w_i = k_i = \lceil \log_2 \mu(p_i) \rceil$ defined by the number of garbage outputs are required to distinguish all occurrences of p_i. Assume that h such leaf vertices are added. After this initialization, we combine the two vertices v_j and v_k ($1 \leq j, k \leq h$) with the smallest weights to a new vertex v_{h+1} with weight $w_{h+1} = \max(w_j, w_k) + 1$. This represents that w_{h+1} outputs are required to distinguish all occurrences of the output patterns p_j and p_k (plus one additional primary output to distinguish the two patterns). These combinations are repeated until a single vertex – the root of the Huffman tree – results.

Example 5 *Consider again the distribution of the output patterns as shown in Table IIb. To determine the Huffman code, we start with the vertices* v_1*,* v_2*,* v_3*, and* v_4 *– one for each output pattern* p_i *with* $\mu(p_i) > 0$*. These vertices are shown at the bottom of Fig. 2. The weights are drawn inside the respective vertices. The weight of vertex* v_1 *is* $w_1 = k_1 = 2$*, because output pattern* $p_1 = 010$ *requires two garbage outputs. The weights of the vertices representing* p_2*,* p_3*, and* p_4 *are 1, 0, and 0, respectively. In a first step, we combine the vertices* v_3 *and* v_4 *(both have weight 0). The resulting vertex* v_5 *has a weight of* $w_5 = \max(0, 0) + 1 = 1$*. Next, we combine the two vertices with weight 1 (i.e.* v_2 *and* v_5*). The resulting vertex* v_6 *has a weight of* $w_6 = \max(1, 1) + 1 = 2$*. Finally, the two remaining vertices are combined to a new vertex* v_7 *with weight* $w_7 = \max(2, 2) + 1 = 3$ *– eventually resulting in the tree shown in Fig. 2.*

After generating the Huffman tree, the overall number of variables that are required to realize the encoded function in reversible logic is given by the weight of the root vertex of the tree. The resulting code is inherently given by the structure of the Huffman tree. In fact, each path from the root vertex to a leaf vertex represents a code word, where taking the left (right) edge implies a 0 (1).

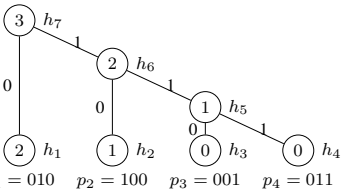

Fig. 2. Huffman tree for the function from Table IIa

Example 5 (continued) *Since the root vertex has a weight of 3, three variables are required to realize the encoded function (note again that, without encoding,* $\max(3, 3+2) = 5$ *variables would be required). The path from the root vertex to the leaf vertex* h_2 *(which represents output pattern* p_2*) traverses the right edge of the root vertex* h_7 *as well as the left edge of* h_6*. Consequently,* $c(p_2) = 10$ *encodes* $p_2 = 100$*. Since* h_2 *has weight* $w_2 = 1$*, one output is used as garbage output in this case. Accordingly, code words for all other output patterns are determined – eventually resulting in the code* c *shown in Table IIIa. Dashes again represent don't cares.*

Note that the number of output patterns that may occur at least once may increase exponentially with the total number of inputs – a significant threat to the proposed approach with respect to scalability. However, experimental evaluations (later summarized in Section V) shows that, for most functions, the number of output patterns p_i with $\mu(p_i) > 0$ is feasible (the vast majority of output patterns never occur and, hence, do not need to be considered in the code).

B. Exploiting the Encoding During Synthesis

In contrast to the established design process reviewed in Section III.A, the Huffman code yields a function to be synthesized in which single outputs may be used as both, actual primary outputs as well as garbage outputs – dependent on the output pattern. Besides the fact that this allows for performing the actual synthesis with a significantly smaller number of variables, it inherently motivates a synthesis scheme in which a huge degree of freedom can be exploited. More precisely, the *don't care* values can arbitrarily be assigned since their value does not matter as long as a reversible function results. Since this is an inherently given property of reversible circuits, we do not have to care about their value. To show how these *don't cares* can be exploited to reduce the complexity of the resulting circuits, we briefly discuss the main concepts behind functional synthesis approaches first.

In general, functional synthesis approaches operate on a function description F of the function to be synthesized (e.g. a permutation matrix). The goal is to determine a sequence of gates G that transforms the function description to the identity. Since $F \circ F^{-1} = I$, this yields a reversible circuit G that realizes F^{-1}. From this, we can easily obtain a realization for F by reversing the order of the gate in G and exchanging each gate with its inverse (Toffoli gates are self-inverse).

Since we aim for exploiting the don't care values, we propose the strategy to transform one variable after the other to the identity – starting at the most significant variable. To this end, we represent the function to be synthesized using a permutation matrix. This is motivated by the fact that compact representations for permutation matrices exists (e.g. QMDDs as proposed in [15]).

Example 6 *Applying the determined Huffman code (cf. Table IIIa) to the output patterns of the function shown in Table IIa yields the encoded function shown in Table IIIb. The corresponding permutation matrix is depicted in Fig. 3a. In the original function, the input combinations 000, 001, 101, and 110 are all mapped to the output patterns* $p_1 = 010$*. Since this output pattern is now encoded by 0--, there are four possibilities for each of the input combinations where to locate the corresponding 1-entry in the permutation matrix (the only requirement is that the 1-entries must be in different rows to guarantee reversibility). This degree of freedom is sketched by a blue rectangle in Fig. 3. A similar degree of freedom (sketched by a red rectangle in Fig. 3) occurs for output pattern 100 which is encoded by 10-.*

As mentioned above, we aim for transforming the permutation matrix to the identity by transforming one variable after the other to the identity – starting with the most significant variable x_n. Recall, that a variable represents a mapping from one input bit to the corresponding output bit. Obviously, there exist four possibilities of such a mapping: 0 is mapped to 0, 0 is mapped to 1, 1 is mapped to 0, or 1 is mapped to 1. For the most significant variable x_n, these possibilities are represented by the four quadrants of the matrix, e.g. a 1-entry in the top right quadrant represents a mapping of x_n from 1 to 0 for a certain input combination. Since we aim for transforming the permutation matrix to the identity, one has to move all 1-entries from the top right and the bottom left quadrant to the top left and the bottom right quadrant, respectively (obtaining the identity, i.e. a mapping from 0 to 0 or from 1 to 1 for x_n in all cases). Moving 1-entries can be performed by applying Toffoli gates, which allow to swap columns. More precisely, a Toffoli gate $TOF(C_i, t_i)$ swaps the columns represent by $C_i \cup t_i^-$ with those represented by $C_i \cup t_i^+$.

Example 6 (continued) *To establish the identity for the most significant variable* x_3 *in the permutation matrix shown in Fig. 3a, we have to swap the columns 101 and 110 with columns 010 and 011, respectively. This can e.g be achieved by applying the Toffoli gates* $TOF(\{x_1^+, x_3^+\}, x_2)$ *and* $TOF(\{x_2^+\}, x_3)$*. The first gate exchanges columns 101 and 111 (by inverting* x_2*), whereas the second Toffoli gate simultaneously swaps columns 110 and 111 with columns 010 and 011. The resulting permutation matrix is shown in Fig. 3b.*

Once we have established the identity for the most significant variable, we apply this scheme to the next significant variable – affecting the top left and the bottom right quadrant. To ensure that the gates do only affect the currently considered sub-matrix, a negative control line x_n^- is added to each gate that shall be applied to transform the top left quadrant (since the most frequent variable x_n is 0 for all columns of this sub-matrix). Analogously, a positive control line x_n^+ is added to each gate that shall be applied to transform the bottom right quadrant. This is recursively applied for all remaining variables until we reach the least significant one. Hence, the deeper we step into the recursion, the more additional control lines are added to the gates that are required to transform the corresponding sub-matrix to the identity. Even though the number of these additional control lines can be reduced by exploiting some redundancy in the permutation matrix (cf. [19, 27]), this usually yields rather expensive circuits (as reviewed in Section II, the respective costs depend on the number of control lines in each gate).

However, by exploiting the available degree of freedom (provided by the *don't care* values), this overhead can significantly be reduced: Assume without the loss of generality that we have just transformed the most significant variable x_n to the identity and that all other variables $x_{n-1} \dots x_1$ are don't care in case $x_n = 0$ (i.e. the top left quadrant is don't care). This would be the case when the most frequent output pattern p_1 is encoded with $c(p_1) = 0$. Since all values in the top left quadrant are *don't care*, it does not matter which gates are applied to that quadrant. The only constraint is that a reversible function results, which is inherently given since we only apply reversible gates. Consequently, we do not have to care whether the gates that are applied to transform the bottom right quadrant also affects the top left quadrant. Therefore, we do not have to add the additional control line.

Example 6 (continued) *Since the left upper quadrant is composed of* don't care *values only, no positive control line* x_3 *has to be added when applying the algorithm recursively to the bottom right quadrant. Transforming the most significant variable of this sub-matrix (i.e.* x_2*) to the identity requires to apply a Toffoli gate* $TOF(\emptyset, x_2)$*. The resulting permutation matrix is shown in Fig. 3c. Since again the top left quadrant is composed of* don't cares*, no additional control lines are required when applying the algorithm recursively to the* 2×2 *sub-matrix in the*

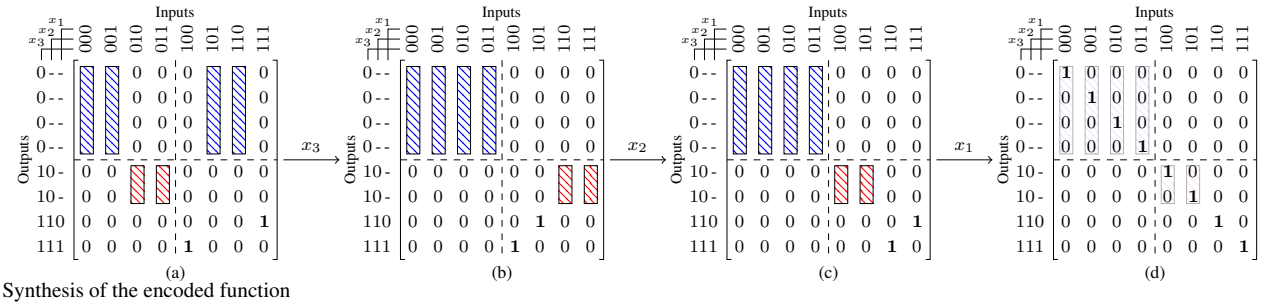

Fig. 3. Synthesis of the encoded function

bottom right corner of the permutation matrix. Finally, the Toffoli gate $TOF(\emptyset, x_1)$ is required to transform this sub-matrix to the identity – eventually resulting in the permutation matrix shown in Fig. 3d.[2] *The resulting circuit (composed of three circuit lines) is shown in the left part of Fig. 4.*

C. Generating the Decoder

As mentioned above, we have used coding techniques in order to allow for a reduction of the variables that have to be considered during synthesis. However, since this realizes a reversible function which is different to the desired one, a decoder is required which recalculates the original outputs. In this decoder, we have to add the variables that were saved during the actual synthesis again – yielding a circuit with a total number of $\max(n, m + k_1)$ inputs/outputs.

A large portion of the decoder can be realized very efficiently: Assume that r additional variables are required such that the overall number of variables is again $\max(n, m + k_1)$. Then, we add r ancillary lines $a_r, \ldots a_1$ (initialized with zero) to the circuit that realizes the encoded function. We use these r lines to decode the r most significant bits of the original output patterns (i.e. $x'_m, \ldots x'_{m-r+1}$). To this end, we traverse all codewords $c(p_i)$. If the codeword encodes an output patterns p_i with $x'_j = 1$ $(m - r < j \leq m)$, we have to invert the value of circuit line $a_{j-(m-r)}$ for that codeword. To this end, we apply a Toffoli gate with target line $a_{j-(m-r)}$ and a set of control lines that represent codeword $c(p_i)$. More precisely, the set of control lines contains a positive (negative) control line for each 0 (1) in $c(p_i)$. This procedure allows to realize r primary outputs very efficiently. An example demonstrates the idea.

Example 7 *Recall the non-reversible function shown in Table IIa and the resulting Huffman code for the output patterns shown in Table IIIa. The left part of Fig. 4 shows the realization of the encoded function (cf. Table IIIb). Since the original function requires $\max(3, 3 + 2) = 5$ variables, we add $r = 2$ further circuit lines (a_2 and a_1) to the circuit. We use these lines to decode the primary outputs x'_3 and x'_2. As discussed above, we traverse the codewords shown in Table IIIa. Codeword $c(p_1) = 0$ encodes output patterns $p_1 = x'_3x'_2x'_1 = 010$. To decode $x'_2 = 1$ for output pattern p_1, we add a Toffoli gate $TOF(\{x_3^-\}, a_1)$ to the circuit. Analogously, we apply gates $TOF(\{x_3^+, x_2^-\}, a_2)$ and $TOF(\{x_3^+, x_2^+, x_1^+\}, a_1)$ to decode codewords $c(p_2) = 10$ and $c(p_4) = 111$ (cf. right part of Fig. 4).*

Note that this procedure cannot be applied to the remaining $m - r$ the primary outputs, because these outputs have to be decoded on lines that are not initialized with constant zero. Hence, these remaining primary outputs have to be decoded using any functional synthesis algorithm (no embedding is required since the function is already reversible). However, since this is necessary only for a small number of primary outputs, it hardly affects the overall synthesis result. This is also confirmed by the experimental evaluation.

[2]The *don't care* values are assigned along the main diagonal. Therefore, no further gates are required.

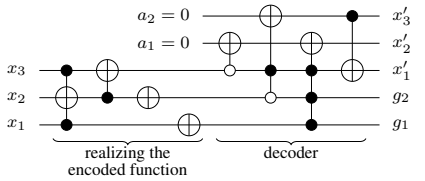

Fig. 4. Resulting circuit

Example 7 *Finally, we have to adjust the least significant primary output x'_1. Note that for all code words except for $c(p_2) = 10$ the most significant bit of the code word is equal to x'_1 of the original output pattern. To establish the desired mapping also for this case, we add a Toffoli gate $TOF(\{a_2\}, x_3)$ – eventually resulting in the circuit shown in Fig. 4. The first four gates of the circuit realizes the encoded function and, thus, operates on three lines only. The remaining gates realize the decoder and, therefore, may operate on all five circuit lines.*

V. Experimental Evaluation

In this section we compare the proposed synthesis flow for functional reversible circuit synthesis to the current state of the art. To this end, we have implemented the proposed approach in C++ utilizing RevKit [17], the QMDD package provided with [15], and the BDD package CUDD [21]. We compared the realizations generated by the proposed approach to two state-of-the-art functional synthesis approaches. The first state-of-the-art approach (denoted $sota_A$ in the following) is composed of the embedding proposed in [20] and a symbolic variant of transformation-based synthesis [16] (executable in RevKit using the command *embed -b; tbs -b*). The second state-of-the-art approach (denoted $sota_B$ in the following) is composed of the embedding proposed in [28] and QMDD-based synthesis (initially proposed in [19] and recently improved in [27]). As benchmarks served functions available at RevLib [24], ISCAS [1], and IWLS93 [12].[3] All experiments were conducted at a 4 GHz machine with 32 GB of memory.

Table IV shows the results of our experimental evaluation. The first four columns list the name of the benchmark, its number of primary inputs n and primary outputs m, as well as the number of variables (circuit lines) l that are required to represent the function in reversible logic. The remaining columns are grouped into three parts – one for each state-of-the-art functional synthesis scheme $sota_A$ as well as $sota_B$ and the proposed synthesis approach that exploits coding techniques during synthesis. For each design approach, we list the required time t (including embedding and synthesis) as well as the T-depth of the resulting circuit. Note that all these approaches end up with with circuits that are composed of l lines. However, since the proposed approach performs the synthesis step on fewer variables, we list the number of lines l_c that are required to realize the encoded function as well in this case (we highlighted the cases where $l_c < l$ in bold).

Table IV clearly shows the improved scalability of the proposed method compared the state-of-the-art approaches. Both state-of-the-art approaches $sota_A$ and $sota_B$ (which are composed of an embedding step followed by the actual synthesis) run

[3]We neglected benchmarks that already describe a reversible function, since in these cases a code cannot be beneficial.

TABLE IV
EXPERIMENTAL RESULTS

				$sota_A$ [20, 16]		$sota_B$ [28, 19, 27]		Proposed		
Name	n	m	l	t	T-depth	t	T-depth	l_c	t	T-depth
9symml	9	1	10	2.02	99 381	0.10	196 764	10	0.07	7 320
life	9	1	10	1.91	100 227	0.08	183 264	10	0.10	5 580
max46	9	1	10	2.00	98 289	0.09	84 000	10	0.06	6 096
sym9	9	1	10	1.90	99 381	0.10	196 764	10	0.07	7 320
clip	9	5	11	2.38	102 489	0.84	581 007	**10**	1.16	207 903
dk27	9	9	15	3.86	123 276	0.89	2 409 495	**10**	0.17	48 405
apex4	9	19	26	26.04	170 100	14.79	14 682 966	**10**	24.73	363 045
sym10	10	1	11	8.04	238 674	0.13	396 120	11	0.12	13 080
sao2	10	4	14	16.19	296 841	1.45	2 189 838	**11**	0.20	32 313
alu2	10	6	14	16.64	308 214	2.58	3 396 543	**11**	1.90	349 125
example2	10	6	14	16.21	308 214	2.63	3 396 543	**11**	1.72	349 125
x2	10	7	16	25.10	391 404	1.98	4 516 011	**11**	0.13	21 075
alu3	10	8	14	19.75	337 281	2.08	3 368 610	**11**	2.51	533 685
ex1010	10	10	18	54.41	475 536	10.10	14 638 416	**11**	7.81	655 830
dk17	10	11	19	56.66	492 033	17.52	37 365 105	**11**	0.94	258 510
apla	10	12	22	199.15	604 542	41.97	77 151 615	**11**	1.00	87 336
cm152a	11	1	11	43.41	638 454	0.02	54 804	11	0.14	143 952
cm85a	11	3	13	54.78	674 883	0.18	948 348	**12**	0.53	123 633
add6	12	7	13	288.91	1 606 533	13.64	5 067 762	13	31.25	5 099 763
alu1	12	8	18	1073.99	2 389 212	7.96	16 141 887	**13**	20.02	2 984 298
co14	14	1	15	TO	–	0.04	26 544	15	0.01	3 360
alu4	14	8	19	TO	–	331.85	324 374 364	**15**	70.39	11 027 733
f51m	14	8	19	TO	–	TO	–	**15**	158.65	15 178 923
tial	14	8	19	TO	–	TO	–	**15**	78.27	10 438 254
cu	14	11	25	TO	–	TO	–	**15**	0.63	76 311
misex3	14	14	28	TO	–	TO	–	**15**	522.65	3 925 806
misex3c	14	14	28	TO	–	TO	–	**15**	540.23	3 956 025
table3	14	14	28	TO	–	TO	–	**15**	6.93	463 260
s1488	14	25	38	TO	–	TO	–	**15**	197.74	9 553 668
s1494	14	25	38	TO	–	TO	–	**15**	207.13	9 180 474
b12.pla	15	9	22	TO	–	51.31	151 591 518	**16**	115.10	17 219 412
in0	15	11	25	TO	–	TO	–	**16**	81.27	11 725 497
parity	16	1	16	2490.59	9 083 781	0.01	0*	16	0.18	0*
ryy6	16	1	17	TO	–	1.27	80 349 048	17	0.03	6 078
t481	16	1	17	TO	–	12.33	89 912 472	17	0.02	288
cmb	16	4	20	TO	–	TO	–	**17**	0.02	9 036
pcler8	16	5	21	TO	–	18.11	60 332 238	**17**	56.69	7 878 141
cm163a	16	13	25	TO	–	TO	–	**17**	708.99	80 405 748
pdc	16	40	55	TO	–	TO	–	**17**	3004.29	10 401 426
spla	16	46	61	TO	–	TO	–	**17**	2488.81	13 852 266
table5	17	15	32	TO	–	TO	–	**18**	77.55	10 065 483
cm150a	21	1	22	TO	–	TO	–	22	0.36	7 704
mux	21	1	22	TO	–	TO	–	22	0.48	7 056
cordic	23	2	25	TO	–	TO	–	**24**	1028.91	17 630 250
e64	65	65	129	TO	–	TO	–	**65**	4.84	95 202

* A *T-depth* of zero indicates that the resulting circuit is soely composed of Toffoli gates with at most one control line.

into the timeout of 1 hour in several cases, whereas the proposed synthesis approach allows for synthesizing a reversible circuit within a second (e.g. for benchmarks *cm150a* and *mux*).[4] This significant improvement can be explained since the synthesis in the proposed synthesis scheme operates on significantly fewer variables. Comparing the columns l_c and l of Table IV shows that, in 32 out of 45 cases, the proposed code allows for a reduction of the variables that have to be considered during synthesis. This allows e.g. to synthesize benchmark *e64*, which is composed of 65 inputs and outputs (and requires 129 circuit lines), within 5 seconds, whereas the previously proposed approaches already timeout in the embedding step.

Besides the significant improvement with respect to scalability, also the complexity of the resulting circuits is significantly reduced. In the majority of the cases, the proposed approach generates significantly smaller circuits (e.g. for benchmarks *apla* or *x2*). Only in rare cases, we obtain circuits that are more complex than those generated by the state-of-the-art flows $sota_A$ and $sota_B$. On average, we obtain a reduction of the *T-depth* of 66.3% compared to the $sota_A$ and a reduction of the *T-depth* of 92.6% compared to $sota_B$.[5] There exist cases (e.g. *ryy6* and *t481*) in which the proposed synthesis approach generates circuits with magnitudes fewer *T-depth* than $sota_B$. These significant reductions in *T-depth* can be explained by the fact that significantly fewer variables were considered during synthesis in many cases – leading to gates with fewer control lines. Furthermore, exploiting *don't cares* during synthesis and the efficient construction of the decoder (cf. Section IV) allowed for realizing the desired functionality in reversible logic using rather few gates.

[4]The largest part of the runtime was consumed is the synthesis of the encoded function. The Huffman code could be generated withing a few seconds for all benchmarks and the decoder could have been synthesized within 100 seconds.

[5]These differences on the average improvements is not surprising, since $sota_A$ typically generates less complex circuits than $sota_B$. However, $sota_B$ is more scalable.

VI. CONCLUSION

In this work, we have shown how coding techniques can be exploited to improve the functional synthesis of reversible circuits. By using a variable-length code for the output patterns, significantly fewer variables have to be considered during synthesis (which is the main limitation of the scalability of current approaches). Besides that, the resulting codes contain a large portion of *don't care* values, which can be exploited during synthesis in order to further reduce the complexity of the resulting circuit. Since this results in a reversible circuit that realizes an encoded function, a decoder has to be applied afterwards (which, however, can again be realized rather efficiently). Experimental evaluations confirm the benefits of this approach: A significant improvement with respect to scalability compared to state-of-the-art approaches has been observed. Besides that, significantly smaller circuit result in the majority of the cases – sometimes circuits with several orders of magnitude fewer quantum cost have been realized. Future work includes a comparison of the proposed synthesis scheme to recently proposed design flows such as one-pass design of reversible circuits [29].

ACKNOWLEDGEMENTS

This work has partially been supported by the European Union through the COST Action IC1405.

REFERENCES

[1] ISCAS'89 benchmark information. *www.cbl.ncsu.edu/ CBL_Docs/ iscas89.html*, 1997.
[2] L. G. Amarù, P. Gaillardon, R. Wille, and G. D. Micheli. Exploiting inherent characteristics of reversible circuits for faster combinational equivalence checking. In *Design, Automation and Test in Europe*, pages 175–180, 2016.
[3] M. Amy, D. Maslov, M. Mosca, and M. Roetteler. A meet-in-the-middle algorithm for fast synthesis of depth-optimal quantum circuits. *IEEE Trans. on CAD of Integrated Circuits and Systems*, 32(6):818–830, 2013.
[4] C. H. Bennett. Logical reversibility of computation. *IBM J. Res. Dev*, 17(6):525–532, 1973.
[5] A. Berut, A. Arakelyan, A. Petrosyan, S. Ciliberto, R. Dillenschneider, and E. Lutz. Experimental verification of Landauer's principle linking information and thermodynamics. *Nature*, 483:187–189, 2012.
[6] K. Fazel, M. Thornton, and J. Rice. ESOP-based Toffoli gate cascade generation. In *Communications, Computers and Signal Processing, 2007. PacRim 2007. IEEE Pacific Rim Conference on*, pages 206 –209, 2007.
[7] D. Große, R. Wille, G. W. Dueck, and R. Drechsler. Exact multiple control Toffoli network synthesis with SAT techniques. *IEEE Trans. on CAD*, 28(5):703–715, 2009.
[8] P. Gupta, A. Agrawal, and N. K. Jha. An algorithm for synthesis of reversible logic circuits. *IEEE Trans. on CAD*, 25(11):2317–2330, 2006.
[9] D. A. Huffman. A method for the construction of minimum-redundancy codes. *Proceedings of the IRE*, 40(9):1098–1101, 1952.
[10] R. Landauer. Irreversibility and heat generation in the computing process. *IBM Journal of Research and Development*, 5(3):183–191, 1961.
[11] D. Maslov and G. W. Dueck. Reversible cascades with minimal garbage. *IEEE Trans. on CAD*, 23(11):1497–1509, 2004.
[12] K. McElvain. IWLS'93 benchmark set: Version 4.0. In *Int'l Workshop on Logic Synth.*, 1993.
[13] D. M. Miller, D. Maslov, and G. W. Dueck. A transformation based algorithm for reversible logic synthesis. In *Design Automation Conf.*, pages 318–323, 2003.
[14] M. Nielsen and I. Chuang. *Quantum Computation and Quantum Information*. Cambridge Univ. Press, 2000.
[15] P. Niemann, R. Wille, D. M. Miller, M. A. Thornton, and R. Drechsler. QMDDs: Efficient quantum function representation and manipulation. *IEEE Trans. on CAD*, 35(1):86–99, 2016.
[16] M. Soeken, G. W. Dueck, and D. M. Miller. A fast symbolic transformation based algorithm for reversible logic synthesis. In *Reversible Computation - 8th Int'l Conf.*, pages 307–321, 2016.
[17] M. Soeken, S. Frehse, R. Wille, and R. Drechsler. RevKit: A toolkit for reversible circuit design. In *Workshop on Reversible Computation*, pages 69–72, 2010. RevKit is available at http://www.revkit.org.
[18] M. Soeken, L. Tague, G. W. Dueck, and R. Drechsler. Ancilla-free synthesis of large reversible functions using binary decision diagrams. *J. Symb. Comput.*, 73:1–26, 2016.
[19] M. Soeken, R. Wille, C. Hilken, N. Przigoda, and R. Drechsler. Synthesis of reversible circuits with minimal lines for large functions. In *ASP Design Automation Conf.*, pages 85–92, 2012.
[20] M. Soeken, R. Wille, O. Keszocze, D. M. Miller, and R. Drechsler. Embedding of large Boolean functions for reversible logic. *J. Emerg. Technol. Comput. Syst.*, 12(4):41:1–41:26, 2015.
[21] F. Somenzi. CUDD: CU decision diagram package release 3.0. 0. 2015.
[22] R. Wille and R. Drechsler. BDD-based synthesis of reversible logic for large functions. In *Design Automation Conf.*, pages 270–275, 2009.
[23] R. Wille, R. Drechsler, C. Osewold, and A. G. Ortiz. Automatic design of low-power encoders using reversible circuit synthesis. In *Design, Automation and Test in Europe*, pages 1036–1041, 2012.
[24] R. Wille, D. Große, L. Teuber, G. W. Dueck, and R. Drechsler. RevLib: an online resource for reversible functions and reversible circuits. In *Int'l Symp. on Multi-Valued Logic*, pages 220–225, 2008. RevLib is available at http://www.revlib.org.
[25] R. Wille, O. Keszöcze, and R. Drechsler. Determining the minimal number of lines for large reversible circuits. In *Design, Automation and Test in Europe*, 2011.
[26] R. Wille, O. Keszocze, S. Hillmich, M. Walter, and A. G. Ortiz. Synthesis of approximate coders for on-chip interconnects using reversible logic. In *Design, Automation and Test in Europe*, 2016.
[27] A. Zulehner and R. Wille. Improving synthesis of reversible circuits: Exploiting redundancies in paths and nodes of QMDDs. In *Conf. on Reversible Computation*, 2017.
[28] A. Zulehner and R. Wille. Make it reversible: Efficient embedding of non-reversible functions. In *Design, Automation and Test in Europe*, 2017.
[29] A. Zulehner and R. Wille. One-pass design of reversible circuits: Combining embedding and synthesis for reversible logic. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 2017.
[30] A. Zulehner and R. Wille. Taking one-to-one mappings for granted: Advanced logic design of encoder circuits. In *Design, Automation and Test in Europe*, 2017.