LUT-Based Hierarchical Reversible Logic Synthesis

Mathias Soeken, *Member, IEEE*, Martin Roetteler, *Member, IEEE*, Nathan Wiebe, and Giovanni De Micheli, *Fellow, IEEE*

***Abstract*—We present a synthesis framework to map logic networks into quantum circuits for quantum computing. The synthesis framework is based on lookup-table (LUT) networks, which play a key role in conventional logic synthesis. Establishing a connection between LUTs in an LUT network and reversible single-target gates in a reversible network allows us to bridge conventional logic synthesis with logic synthesis for quantum computing, despite several fundamental differences. We call our synthesis framework LUT-based hierarchical reversible logic synthesis (LHRS). Input to LHRS is a classical logic network representing an arbitrary Boolean combinational operation; output is a quantum network (realized in terms of Clifford+*T* gates). The framework allows one to account for qubit count requirements imposed by the overlying quantum algorithm or target quantum computing hardware. In a fast first step, an initial network is derived that only consists of single-target gates and already completely determines the number of qubits in the final quantum network. Different methods are then used to map each single-target gate into Clifford+*T* gates, while aiming at optimally using available resources. We demonstrate the versatility of our method by conducting a design space exploration using different parameters on a set of large combinational benchmarks. On the same benchmarks, we show that our approach can advance over the state-of-the-art hierarchical reversible logic synthesis algorithms.**

***Index Terms*—Combinational circuits, design automation, quantum computing.**

I. Introduction

RECENT progress in fabrication makes the practical application of quantum computers a tangible prospect [2]–[5]. However, as quantum computers scale up to tackle problems in computational chemistry, machine learning, and cryptoanalysis, design automation will be necessary to fully leverage the power of this emerging computational model.

Quantum circuits differ significantly in comparison to classical circuits. This needs to be addressed by design automation tools.

1) Quantum computers process *qubits* instead of classical bits. A qubit can be in superposition and several qubits can be entangled. We target purely Boolean functions as input to our synthesis algorithms. At design time, it is sufficient to assume that all input values are Boolean, even though entangled qubits in superposition are eventually acted upon by the quantum hardware.
2) All operations on qubits besides measurement, called quantum gates, must be *reversible*. Gates with multiple fanout known from classical circuits are therefore not possible. Temporarily computed values must be stored on additional helper qubits, called ancillae. An intensive use of intermediate results therefore increases the qubit requirements of the resulting quantum circuit. Qubits are a limited resource; therefore, the use of ancillae is restricted and synthesis must find circuits that satisfy the number of available qubits. Quantum circuits that compute a purely Boolean function are often referred to as reversible networks.
3) The quantum gates that can be implemented by current quantum computers can act on a single or at most two qubits [3]. Something as simple as an AND operation can therefore not be expressed by a single quantum gate. A universal fault-tolerant quantum gate library is the Clifford+*T* gate set [3]. In this gate set, the *T* gate is sufficiently expensive in most approaches to fault tolerant quantum computing such that it is customary to neglect all other gates when costing a quantum circuit [6]. Mapping reversible functions into networks that minimize *T* gates is therefore a central challenge in quantum computing [7].
4) When executing a quantum circuit on a quantum computer, all qubits must eventually hold either a primary input value, a primary output value, or a constant. A circuit should not expose intermediate results to output lines as this can potentially destroy wanted interference effects, in particular if the circuit is used as a subroutine in a larger quantum computation. Qubits that nevertheless expose intermediate results are sometimes referred to as garbage outputs.

It has recently been shown [8]–[10] that hierarchical reversible logic synthesis methods based on logic network representations are able to synthesize large arithmetic designs. The underlying idea is to map subnetworks into reversible networks. *Hierarchical* refers to the property that intermediate results computed by the subnetworks must be stored on additional ancilla qubits. If the subnetworks are small enough, one can locally apply less efficient reversible synthesis methods that do not require ancilla qubits and are based on Boolean satisfiability [11], truth tables [12], or decision diagrams [13]. However, state-of-the-art hierarchical synthesis methods mainly suffer from two disadvantages. First, they do not explicitly *uncompute* the temporary values from the subnetworks and leave garbage outputs. In order to use the network in a quantum computer, one can apply a technique

Manuscript received June 21, 2017; revised November 11, 2017, March 2, 2018, and May 20, 2018; accepted June 23, 2018. Date of publication July 24, 2018; date of current version August 20, 2019. This work was supported in part by CyberCare under Grant H2020-ERC-2014-ADG 669354, in part by the Swiss National Science Foundation under Grant 200021-169084 MAJesty, and in part by ICT COST Action under Grant IC1405. A preliminary version of this manuscript has been presented at the DAC 2017 conference [1]. This paper was recommended by Associate Editor S.-C. Chang. *(Corresponding author: Mathias Soeken.)*

M. Soeken and G. De Micheli are with the Integrated Systems Laboratory, EPFL, 1015 Lausanne, Switzerland (e-mail: mathias.soeken@epfl.ch).

M. Roetteler and N. Wiebe are with Microsoft Research, Redmond, WA 98052 USA.

Digital Object Identifier 10.1109/TCAD.2018.2859251

called "Bennett trick" [14], which requires to double the number of gates and add one further ancilla for each primary output. Second, current algorithms do not offer satisfying solutions to account for qubit limits. Either qubit and gate count optimization is intertwined in a single step (see [9], [10], [15]) or applied as a post-optimization step [16].

In order to address these two disadvantages, in this paper we present a hierarchical synthesis framework based on k-feasible Boolean logic networks, which find use in conventional logic synthesis. These are logic networks in which every gate, called LUT, has at most k inputs. They are often referred to as k-LUT (lookup table) networks. Our main contributions are as follows.

1) We point out a one-to-one correspondence between a k-LUT in a logic network and a reversible single-target gate with k control lines in a reversible network. A single-target gate has a k-input control function and a single target line that is inverted if and only if the control function evaluates to 1 (the initial idea for this correspondence and a preliminary evaluation has been presented before in [1]).
2) We describe a two-stage algorithm which first maps a k-LUT network into a reversible single-target gate network and then each single-target gate into quantum circuits. The first step is used to quickly derive a single-target gate network which provides a skeleton for subsequent synthesis that already fixes the number of qubits in the final quantum network. This decouples qubit count optimization from gate count optimization, which makes it easier to respect qubit restrictions imposed by the quantum algorithm or target quantum computing device.
3) We propose new and different methods for the second step, which map each single-target gate into a Clifford+T network. A direct method, introduced in [1], makes use of the exclusive-sum-of-product (ESOP) representation of the control function that can be directly translated into multiple-controlled Toffoli gates [17]. Multiple-controlled Toffoli gates are a specialization of single-target gates for which automated translations into Clifford+T circuits exist. Another method tries to remap a single-target gate into an LUT network with fewer number of inputs in the LUTs, by making use of temporarily unused qubits in the overall quantum network. We show that near-optimal Clifford+T circuits can be precomputed and stored in a database if such LUT networks require sufficiently few gates.

We evaluated LHRS on the EPFL arithmetic combinational benchmarks. The experiments show how the various synthesis parameters effect the number of qubits and the number of T gates in the final quantum network as well as the algorithm's runtime. We also show that the synthesis results can significantly improve state-of-the-art results, particularly when comparing the number of required qubits. Although some of the synthesized benchmarks still require a large amount of resources both in qubits and gate count, our proposed framework offers a new way to address qubit and gate count optimization. Quantum programming frameworks [18], such as Q#, LIQ$Ui|\rangle$ [19], and ProjectQ [20] can link in the Clifford+T circuits that are automatically generated by LHRS.

II. Preliminaries

A. Some Notation

A digraph $G = (V, A)$ is called *simple*, if $A \subseteq V \times V$, i.e., there can be at most one arc between two vertices for each direction. An acyclic digraph is called a *dag*. We refer to $d^-(v) = \#\{w \mid (w, v) \in A\}$ and $d^+(v) = \#\{w \mid (v, w) \in A\}$ as *in-degree* and *out-degree* of v, respectively.

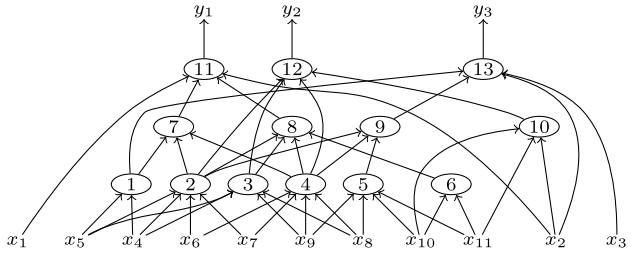

Fig. 1. Four-feasible network with 11 inputs, 3 outputs, and 13 gates.

B. Boolean Logic Networks

A Boolean *logic network* is a simple dag whose vertices are primary inputs, primary outputs, and gates and whose arcs connect gates to inputs, outputs, and other gates. Formally, a Boolean logic network $N = (V, A, F)$ consists of a simple dag (V, A) and a function mapping F. It has vertices $V = X \cup Y \cup G$ for primary inputs X, primary outputs Y, and gates G. We have $d^-(x) = 0$ for all $x \in X$ and $d^+(y) = 0$ for all $y \in Y$. Arcs $A \subseteq (X \cup G \times G \cup Y)$ connect primary inputs and gates to other gates and primary outputs. Each gate $g \in G$ realizes a Boolean function $F(g) : \mathbb{B}^{d^-(g)} \to \mathbb{B}$, i.e., the number of inputs in $F(g)$ coincides with the number of ingoing arcs of g.

Example 1: Fig. 1 shows a logic network of the benchmark *cm85a* obtained using ABC [21]. It has 11 inputs, 3 outputs, and 13 gates. The gate functions are not shown but it can easily be checked that each gate has at most four inputs.

The *fanin* of a gate or output $v \in G \cup Y$, denoted $\text{fanin}(v)$, is the set of source vertices of ingoing arcs

$$\text{fanin}(v) = \{w \mid (w, v) \in A\}. \tag{1}$$

For a gate $g \in G$, this set is ordered according to the position of variables in $F(g)$. For a primary output $y \in Y$, we have $d^-(y) = 1$, i.e., $\text{fanin}(y) = \{v\}$ for some $v \in X \cup G$. The vertex v is called *driver* of y and we introduce the notation $\text{driver}(y) = v$. The *transitive fan-in* of a vertex $v \in V$, denoted $\text{tfi}(v)$, is the set containing v itself, all primary inputs that can be reached from v, and all gates which are on any path from v to the primary inputs. The transitive fan-in can be constructed using the following recursive definition:

$$\text{tfi}(v) = \begin{cases} \{v\} & \text{if } v \in X \\ \{v\} \cup \bigcup\limits_{w \in \text{fanin}(v)} \text{tfi}(w) & \text{otherwise.} \end{cases} \tag{2}$$

Example 2: The transitive fan-in of output y_3 in the logic network in Fig. 1 contains $\{y_3, 1, 2, 4, 5, 9, 13, x_2, x_3, x_4, x_5, x_6, x_7, x_8, x_9, x_{10}, x_{11}\}$. The driver of y_3 is gate 13.

We call a network *k-feasible* if $d^-(g) \leq k$ for all $g \in G$. Sometimes k-feasible networks are referred to as k-LUT networks (LUT is a shorthand for lookup-table) and LUT mapping (see [22]–[26]) refers to a family of algorithms that obtain k-feasible networks, e.g., from homogeneous logic representations, such as And-inverter graphs (AIGs, [27]) or majority-inverter graphs [28].

Example 3: The logic network in Fig. 1 is 4-feasible.

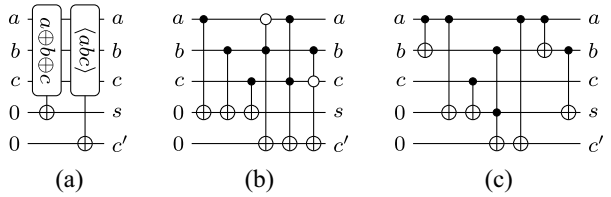

Fig. 2. Reversible circuit for a full adder using (a) two single-target gates, (b) three Toffoli gates and three CNOT gates, and (c) one Toffoli gate and six CNOT gates.

C. Reversible Logic Networks

A reversible logic network realizes a reversible function, which makes it very different as compared to conventional logic networks. The number of lines, which correspond to logical qubits, remains the same for the whole network, such that reversible networks are cascades of reversible gates and each gate is applied to the current qubit assignment. The most general reversible gate we consider in this paper is a *single-target gate*. A single-target gate $T_c(\{x_1, \ldots, x_k\}, x_{k+1})$ has an ordered set of *control lines* $\{x_1, \ldots, x_k\}$, a *target line* x_{k+1}, and a *control function* $c : \mathbb{B}^k \to \mathbb{B}$. It realizes the reversible function $f : \mathbb{B}^{k+1} \to \mathbb{B}^{k+1}$ with $f : x_i \mapsto x_i$ for $i \leq k$ and $f : x_{k+1} \mapsto x_{k+1} \oplus c(x_1, \ldots, x_k)$. It is known that all reversible functions can be realized by cascades of single-target gates [29]. We use the "$\circ$" operator for concatenation of gates.

Example 4: Fig. 2(a) shows a reversible circuit that realizes a full adder using two single-target gates, one for each output. Two additional lines, called *ancilla* and initialized with 0, are added to the network to store the result of the outputs. All inputs are kept as output.

A *multiple-controlled Toffoli gate* is a single-target gate in which the control function is 1 (tautology) or can be expressed in terms of a single product term. One can always decompose a single-target gate $T_c(\{x_1, \ldots, x_k\}, x_{k+1})$ into a cascade of Toffoli gates

$$T_{c_1}(X_1, x_{k+1}) \circ T_{c_2}(X_2, x_{k+1}) \circ \cdots \circ T_{c_l}(X_l, x_{k+1}) \qquad (3)$$

where $c = c_1 \oplus c_2 \oplus \cdots \oplus c_l$, each c_i is a product term or 1, and $X_i \subseteq \{x_1, \ldots, x_k\}$ is the support of c_i. This decomposition of c is also referred to as ESOP decomposition [30]–[32]. ESOP minimization algorithms try to reduce l, i.e., the number of product terms in the ESOP expression. Smaller ESOP expressions lead to fewer multiple-controlled Toffoli gates in the decomposition of a single-target gate. If $c = 1$, we refer to $T_c(\emptyset, x_{k+1})$ as NOT gate, and if $c = x_i$, we refer to $T_c(\{x_i\}, x_{k+1})$ as CNOT gate.

Example 5: Fig. 2(b) shows the full adder circuit from the previous example in terms of Toffoli gates. Each single-target gate is expressed in terms of three Toffoli gates. Positive and negative control lines of the Toffoli gates are drawn as solid and white dots, respectively. Fig. 2(c) shows an alternative realization of the same output function, albeit with one Toffoli gate.

For more details on reversible circuits we refer the reader to [33].

D. Mapping to Quantum Circuits

Quantum circuits are described in terms of a small library of gates that interact with one or two qubits. One of the most frequently considered libraries is the so-called Clifford+T gate library. It consists of the reversible CNOT gate, the Hadamard gate, abbreviated H, as well as the T gate [34], and its inverse $T^\dagger = T^{-1} = T^7$. Quantum gates on n qubits are represented as $2^n \times 2^n$ unitary matrices. We write $T^\dagger$ to mean the complex conjugate of T, and use the symbol "†" also for other quantum gates. The T gate is sufficiently expensive in most approaches to fault tolerant quantum computing [6] that it is reasonable to only consider the T gate when costing a quantum algorithm. For more details on quantum gates we refer the reader to [34].

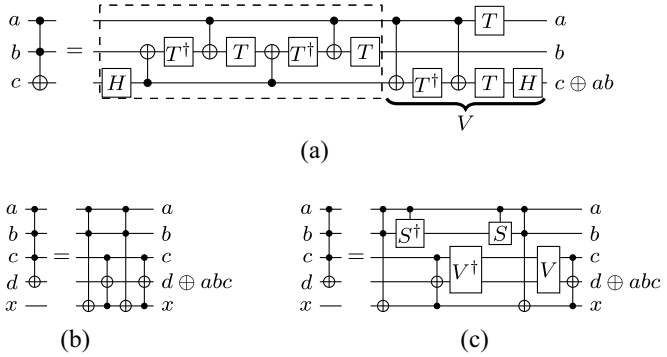

Fig. 3. Mapping Toffoli gates into Clifford+T circuits. (a) Two-controlled Toffoli gate. (b) Three-controlled Toffoli gate (28 T gates). (c) Three-controlled Toffoli gate (16 T gates).

Fig. 3(a) shows one of the many realizations of the two-controlled Toffoli gate, which can be found in [7]. It requires seven T gates which is optimum [6], [35]. Several works from the literature describe how to map larger multiple-controlled Toffoli gates into Clifford+T gates (see [6], [7], [36], [37]). Fig. 3(b) shows one way to map the three-controlled Toffoli gate using a direct method as proposed by Barenco *et al.* [38] Given a free ancilla line (that does not need to be initialized to 0), it allows to map any multiple-controlled Toffoli gate into a sequence of two-controlled Toffoli gates which can then each be mapped into the optimum network with T-count 7. However, the number of T gates can be reduced by modifying the Toffoli gates slightly. It can be easily seen that the network in Fig. 3(c) is the same as in Fig. 3(b), since the controlled $S^\dagger$ gate cancels the controlled S gate and the $V^\dagger$ gate cancels the V gate. However, the Toffoli gate combined with a controlled S gate can be realized using only four T gates [36], and applying the V to the Clifford+T realization cancels another three T gates [see Fig. 3(a) and [7], [39]]. In general, a k-controlled Toffoli gate can be realized with at most $16(k-1)$ T gates. If the number of ancilla lines is larger or equal to $\lfloor[(k-1)/2]\rfloor$, then $8(k-1)$ T gates suffice [7], [38]. Future improvements to the decomposition of multiple-controlled Toffoli gates into Clifford+T circuits will have an immediate positive effect on our proposed synthesis method.

III. Motivation and Problem Definition

In this paper we address the following problem: given a conventional combinational logic network that represents a desired target functionality and a given number of qubits, find a quantum circuit that does not exceed the number of available qubits and minimizes the number of T gates. The algorithm should be highly configurable such that instead of a single quantum circuit a whole design space of circuits with several Pareto-optimal solutions can be explored.

Algorithm 1: Overview of the LHRS Algorithm

Input : Logic network N, parameters p_{Q}, parameters p_{T}
Output : Clifford+T network R
1 set $N \leftarrow$ lut_mapping(N, p_{Q});
2 set $R \leftarrow$ synthesize_mapping(N, p_{Q});
3 set $R \leftarrow$ synthesize_gates(R, p_{T});
4 **return** R;

A. Algorithm Outline

Algorithm 1 illustrates the general outline of the procedure. The following sections provide further details. Input to the algorithm is a logic network N and it outputs a Clifford+T quantum network R. In addition to N, two sets of parameters p_Q and p_T are provided that control detailed behavior of the algorithm. The parameters will be introduced in the following sections and are summarized in Section VI-A; but for now it is sufficient to emphasize the role of the parameters. Parameters in p_Q can influence both the number of qubits and T gates in R, however, their main purpose is to control the number of qubits. Parameters in p_T only affect the number of T gates.

The first step in Algorithm 1 derives an LUT mapping from the input logic network. One parameter in p_Q is the LUT size for the mapping which has the strongest influence on the number of qubits in R. Given the LUT mapping, one can derive a reversible logic network in which each LUT is translated into one or two single-target gates. In the last step, each of the gates is mapped into Clifford+T gates (Section V).

It is important to know that most of the runtime is consumed by the last step in Algorithm 1, and that after the first two steps the number of qubits for the final Clifford+T network is already known. This allows us to use the algorithm in an incremental way as follows. First, one explores assignments to parameters in p_Q that lead to a desired number of qubits, particularly by evaluating different LUT sizes. This can be done by calling the first two steps of the algorithm with different values for the parameters in p_Q. One can always find a network with $n + m$ qubits, where n is the number of inputs and m is the number of outputs, by setting the LUT size to n. However, the runtime and memory requirements for the second step can become tremendously large in this case. Once a network with a desired number of qubits was found, one can evaluate different mapping strategies for the single-target gates to optimize for the number of T gates by calling the last step by sampling the parameters for p_T. Depending on the parameters used in the first step, this step can take a large fraction of the overall runtime.

IV. Synthesizing the Mapping

This section describes how an LUT mapping can be translated into a reversible network. This is the second step of Algorithm 1. The first step in Algorithm 1 applies conventional LUT mapping algorithms and is not further explained in this paper. The interested reader is referred to [22]–[26].

A. Mapping k-LUTs Into Single-Target Gates

Fig. 4 illustrates the general idea how k-LUT networks are mapped into reversible logic networks composed of single-target gates with control functions with up to k variables.

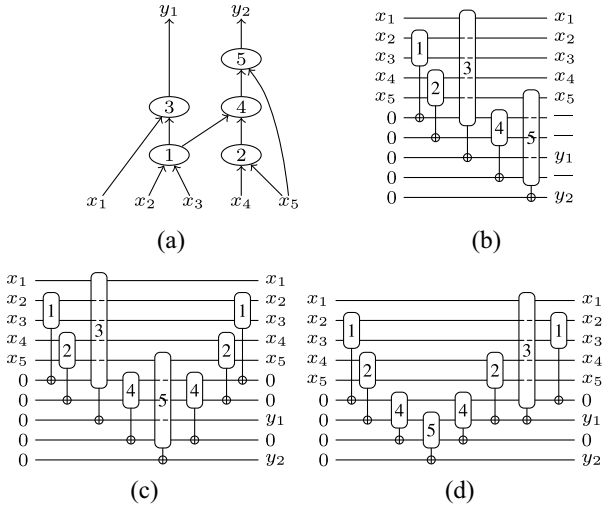

Fig. 4. Simple LUT network to illustrate order heuristics (dashed lines in the single-target gates mean that the line is not input to the gate). (a) LUT network. (b) Reversible network. (c) Order: 1, 2, 3, 4, 5. (d) Order: 1, 2, 4, 5, 3.

Fig. 4(a) shows a two-LUT network with five inputs $x_1, \ldots, x_5$ and five LUTs with names 1 to 5. It has two outputs, y_1 and y_2, which functions are computed by LUT 3 and LUT 5, respectively.

A straightforward way to translate the LUT network is by using one single-target gate for each LUT in topological order. The target of each single-target gate is a 0-initialized new ancilla line. The reversible circuit in Fig. 4(b) results when applying such a procedure. With these five gates, the outputs y_1 and y_2 are realized at lines 8 and 10 of the reversible circuit. But after these first five gates, the reversible circuit has garbage outputs on lines 6, 7, and 9, indicated by "—," which compute the functions of the inner LUTs of the network. The circuit must be free of garbage outputs in order to be implemented on a quantum computer. This is because the result of the calculation is entangled with the intermediate results and so they cannot be discarded and recycled without damaging the results they are entangled with [34]. To avoid the garbage outputs, we can *uncompute* the intermediate results by reapplying the single-target gates for the LUTs in reverse topological order. This disentangles the qubits, reverting them all to constant 0s. Fig. 4(c) shows the complete reversible circuit; the last three gates uncompute intermediate results at lines 6, 7, and 9.

But we can do better! Once we have computed the LUT for a primary output that does not fan in to another LUT, we can uncompute LUTs that are not used any longer by other outputs. The uncomputed lines restore a 0 that can be used to store the intermediate results of other LUTs instead of creating a new ancilla. For the running example, as shown in Fig. 4(d), we can first compute output y_2 and then uncompute LUTs 4 and 2, as they are not in the logic cone of output y_1. The freed ancilla can be used for the single-target gate realizing LUT 3. Compared to the reversible network in Fig. 4(c), this network requires one qubit less by having the exact same gates.

B. Bounds on the Number of Ancillae

As we have seen in the previous section, the order in which LUTs are traversed in the LUT network and translated into

single-target gates affects the number of qubits. Two questions arise: 1) how many ancillae do we need at least and at most and 2) what is a good strategy? We will answer the first question, and then discuss the second one. The bounds assume a given fixed LUT network and a mapping strategy in which each LUT is mapped to at most two single-target gates.

The example order that was used in the previous example leading to the network in Fig. 4(c) illustrates an upper bound. We can always use one ancilla for each LUT in the LUT network, postulated in the following lemma.

Lemma 1: When realizing an LUT network $N = (X \cup Y \cup G, A, F)$ by a reversible circuit that uses single-target gates for each LUT, one needs at most $|G|$ ancilla lines.

The optimized order in Fig. 4(d) used the fact, that one can uncompute gates in the transitive fan-in cone of an output, once the output has been computed.

This observation leads to a lemma providing a lower bound.

Lemma 2: Given an LUT network $N = (X \cup Y \cup G, A, F)$, let $l = \max\{\#\,\mathrm{tfi}(y) \mid y \in Y\}$ be the maximum cone size over all outputs. When realizing the LUT network by a reversible circuit that uses single-target gates for each LUT, we need at least l ancilla lines.

The lower bound inspires the following synthesis strategy that minimizes the number of additional lines. One starts by synthesizing a circuit for the output with the maximum cone. Let us assume that this cone contains l LUTs. These LUTs can be synthesized using l single-target gates. Note that all of these are in fact needed, because in order to uncompute a gate, the intermediate values of children need to be available. From these l gates, $l-1$ gates can be uncomputed (all except the LUT computing the output), and therefore restores $l-1$ lines which hold a constant 0 value. We can easily see that the exact number of required lines may be a bit larger, since all output values need to be kept. Note that this strategy uncomputes all LUTs in the transitive fan-in cone of an output—even if it is part of a fan-in cone of another output. Therefore, some LUTs will lead to more than two single-target gates in the reversible network.

Note that the lower bound only holds when assuming that each LUT is translated to at most two single-target gates. When relaxing this assumption, one can find mappings that require fewer ancillae. One can find such mappings by playing *reversible pebble games* [40], however, we are not considering them in the scope of this paper. More details on reversible pebble games and how they can be used in reversible logic synthesis can be found in [41]–[44].

Algorithm 2: Synthesizing an LUT Mapping Into a Reversible Network With Single-Target Gates

```
1  function synthesize_mapping(N = (X ∪ Y ∪ G, A, F), p_Q)
2      set R ← empty reversible network;
3      set l ← 1;
4      initialize empty stack C;
5      initialize empty map m;
6      set S ← ∅;
7      set D ← {driver(y) | y ∈ Y};
8      for g ∈ G do set r(g) ← d+(g) − [g ∈ D];
9      for x ∈ X do
10         add input line with name x to R;
11         set m(x) ← l;
12         set l ← l + 1;
13     end
14     for g ∈ topo_order(G, p_Q) do
15         set t ← request_constant();
16         append T_F(g)(m(fanin(g)), t) to R;
17         set m(g) ← t;
18         if r(g) = 0 then
19             set S ← ∅;
20             uncompute_children(g);
21         end
22     end
23     for y ∈ Y do
24         rename output of line m(driver(y)) in R to y;
25     end
26     return R;

27 function request_constant()
28     if C is not empty then
29         return C.pop();
30     else
31         set l ← l + 1;
32         return l;
33     end

34 function uncompute_children(g)
35     for g′ ∈ fanin(g) ∩ G do set r(g′) ← r(g′) − 1;
36     for g′ ∈ fanin(g) such that r(g′) = 0 do
37         uncompute_gate(g′);
38     end

39 function uncompute_gate(g)
40     if g ∈ S then return;
41     if g ∉ D then
42         set t ← m(g);
43         append T_F(g)(m(fanin(g)), m(g)) to R;
44         C.push(t);
45         set m(t) ← 0;
46     set S ← S ∪ {g};
47     uncompute_children(g);
```

C. Synthesizing LUT Network

Algorithm 2 describes in detail how a k-LUT network $N = (X \cup Y \cup G, A, F)$ is mapped into a reversible network R that consists of single-target gates with at most k controls. The main entry point is the function "synthesize_mapping" (line 1). This function keeps track of the current number of lines l, available ancillae in a stack C, an LUT-to-line mapping $m : G \to \mathbb{N}$ that stores which LUT gates are computed on which lines in R, and a visited list S (lines 3–6). The reference counter $r(g)$ checks for each LUT g how often it is required as input to other LUTs. For driving LUTs, stored in D (line 7), the reference counter is decreased by 1. It is initialized with the fan-out size and allows us to check if g is not needed any longer such that it can be uncomputed (line 8). This is the case whenever the reference counter is 0, and therefore the process of uncomputing is triggered by driving LUTs.

Input lines are added to R in lines 9–13. Input vertices are mapped to their line in R using m. In lines 14–22 single-target gates to compute and uncompute LUTs are added to R. Each gate g is visited in topological order (details on "topo_order" follow later). First, a 0-initialized line t is requested (line 15). Either there is one in C or we get a new line by incrementing l. Given t, a single-target gate with control function $F(g)$, controls $m(\mathrm{fanin}(g)) = \{m(g') \mid g' \in \mathrm{fanin}(g)\}$, and target line t is added to R (line 16). The LUT-to-line map is updated according to the newly added gate (line 17). Then, if g is driving an output, i.e., $r(g) = 0$ (line 18), we try to uncompute the children recursively by calling uncompute_children (line 20). In that function, first the reference counter is decremented for each child g' that is not a primary input (line 35).

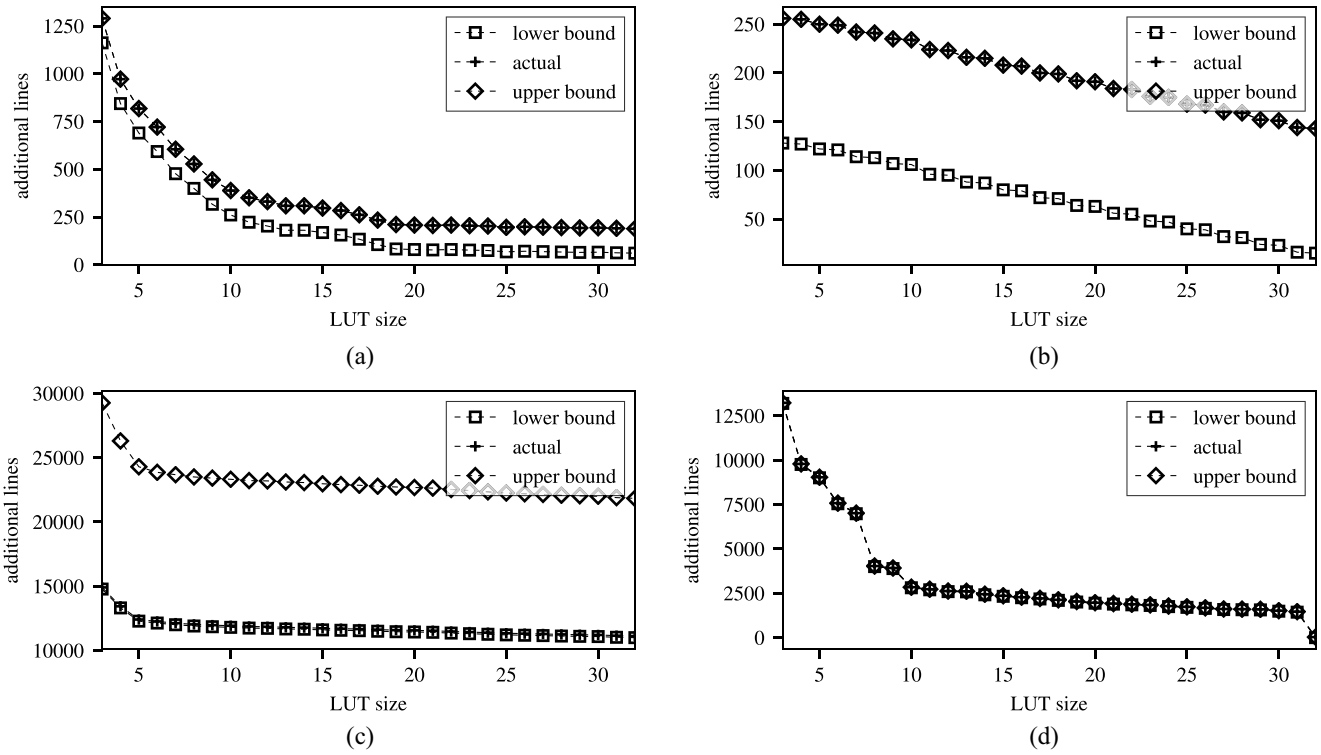

Fig. 5. Plots show the upper and lower bound according to Lemmas 1 and 2 as well as the actual number of additional lines when synthesizing different arithmetic benchmarks with LUT sizes ranging from 3 to 32. The x-axis shows LUT size and the y-axis shows the number of additional lines.

Then, each child g' that afterward has a reference count of 0, is uncomputed using uncompute_gate (line 37). In there, first it is checked whether the child has already been visited (line 40). If the child is not driving an output, a single-target gate to uncompute the line is added to R (line 43). The freed line t is added to C (line 44) and the mapping is cleared accordingly (line 45). The child is stored as visited (line 46), and the function recurs (line 47). After all gates have been computed, outputs are added to lines in R (lines 23–25). This procedure is simplified: two or more outputs may share the same driving LUT. In this case, one needs additional lines and copy the output result using a CNOT gate.

With a given topological order of LUTs, the time complexity of Algorithm 2 is linear in the number of LUTs. As seen in the beginning of this section, the order in which LUTs are visited has an effect on the number of qubits. Therefore, there can be several strategies to compute a topological order on the gates. This is handled by the function topo_order that is configured by a parameter in p_Q. Besides the default topological order implied by the implementation of N (referred to as *topo_def* in the following), we also implemented the order *topo_tfi_sort*, which is inspired by Lemma 2: compute the transitive fan-in cone for each primary output and order them by size in descending order. The topological order is obtained using a depth-first search for each cone by not including duplicates when traversing a cone.

D. Role of the LUT Size

As can be seen from previous discussions, the number of additional lines roughly corresponds to the number of LUTs. Hence, we are interested in logic synthesis algorithms that minimize the number of LUTs. In classical logic synthesis the number of LUT-inputs k needs to be selected according to some target architecture. For example, in field-programmable gate array (FPGA) mapping, its value is typically 6 or 7. But for our algorithm, we can use k as a parameter that trades off the number of qubits to the number of T gates: If k is small, one needs many LUTs to realize the function, but the small number of inputs also limits the number of control lines when mapping the single-target gates into multiple-controlled Toffoli gates. On the contrary, when k is large, one needs fewer LUTs but the resulting Toffoli gates are larger and therefore require more T gates. Further, since for larger k the LUT functions are getting more complex, the runtime to map a single-target gate into multiple-controlled Toffoli gates increases.

To illustrate the influence of the LUT size we performed the following experiment, illustrated in Fig. 5(a). For four benchmarks and for LUT sizes k from 3 to 32, we computed an LUT mapping using ABC's [21] command *if -K k -a*. The resulting network was used to compute both the upper and lower bound on the number of additional lines according to Lemmas 1 and 2, and to compute the actual number of lines according to Algorithm 2 with "topo_def" as topological order. It can be noted that the actual bound often either matches the upper bound or the lower bound. In some cases the bounds are very close to each other, leaving not much flexibility to improve on the number of additional lines. Further, after larger LUT sizes the gain in reducing the number of lines decreases when increasing the LUT size. It should be pointed out that for benchmark "Log2" an optimum number of additional lines can be achieved for $k = 32$, because in this case k matches the number of inputs of the function. Consequently, the LUT mapping has as many gates as the number of outputs. In general, by setting k to the number of inputs n of the function,

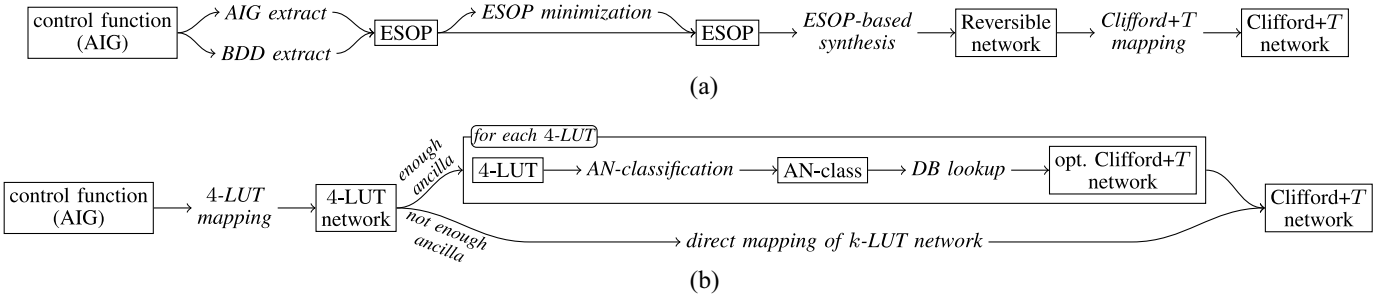

Fig. 6. Algorithms to map a single-target gate into a Clifford+T network. (a) Direct mapping. (b) LUT-based mapping.

one can find reversible single-target gate networks with $n+m$ qubits, where m is the number of outputs. However, first it may be very time consuming or infeasible to map the resulting single-target gates into quantum circuits, and even if it is possible, they will likely consists of a very large number of quantum gates.

V. Mapping Single-Target Gates

For the following discussion it is important to understand the representation of the logic network that is given as input to Algorithm 1 and the k-LUT network that results from the first step. The input network is given as a gate-level logic network, i.e., all gates are simple logic gates. In our experimental evaluation and current implementation the logic network is given as AIG, i.e., a logic network composed of AND gates and inverters. The LUT network is represented by annotating in the gate-level netlist: 1) which nodes are LUT outputs and 2) which nodes are LUT inputs for each LUT. As a result, the control function of a single-target gate corresponds to an LUT, which is *implicitly* represented as a subnetwork in the gate-level logic network.

A. Direct Mapping

The idea of direct mapping is to represent the control function as ESOP expression, optimize it, and then translate the optimized ESOP into a Clifford+T network. Fig. 6(a) illustrates the complete direct mapping flow. As described above, a control function is represented in terms of a multilevel AIG. In order to obtain a 2-level ESOP expression for the control function, one needs to collapse the network. This process is called *cover extraction* and two techniques called *AIG extract* and BDD *extract* will be described in this section. The number of product terms in the resulting ESOP expression is typically far from optimal and is reduced using ESOP minimization. The optimized ESOP expression is first translated into a reversible network with multiple-controlled Toffoli gates as described in Section II-D and then each multiple-controlled Toffoli gate is mapped into a Clifford+T with the mapping described in [7].

1) ESOP Cover Extraction: There are several ways to extract an ESOP expression from an AIG that represents the same function. Our implementation uses two different methods. The choice of the method has an influence on the initial ESOP expression and therefore affects both the runtime of the algorithm and the number of T gates in the final network.

The method *AIG extract* computes an ESOP for each node in the AIG in topological order. The final ESOP can then be read from the output node. First, all primary inputs x_i are assigned the ESOP expression x_i. The ESOP expression of an AND gate is computed by conjoining both ESOP expressions of the children, taking into consideration possible complementation. Therefore, the number of product terms for the AND gate can be as large as the product of the number of terms of the children. The final ESOP can be preoptimized by removing terms that occur twice, e.g., $x_1\bar{x}_2x_3 \oplus x_1\bar{x}_2x_3 = 0$, or by merging terms that differ in a single literal, e.g., $x_1x_3 \oplus x_1\bar{x}_2x_3 = x_1x_2x_3$ [32]. *AIG extract* is implemented similar to the command "&esop" in ABC [21]. We were able to increase the performance of our implementation by limiting the number of inputs to 32 bits, which is sufficient in our application, and by using cube hashing [45]. In general, AIG extract performs linear many conjunctions of ESOP terms, which are quadratic in the size of the ESOP terms of the children nodes. The size of an ESOP term for an AIG node n is exponential in the number of primary inputs in the structural support of n.

The method BDD *extract* first expresses the control function in terms of a BDD (binary descision diagram), again by translating each node into a BDD in topological order. From the BDD a pseudo-Kronecker expression [46], [47] is extracted using the algorithm presented in [48]. A pseudo-Kronecker expression is a special case of an ESOP expression. For the extracted expression, it can be shown that it is minimum in the number of product terms with respect to a chosen variable order. Therefore, it provides a good starting point for ESOP minimization.

For the complexity of BDD extract, we consider both steps. First, the BDD needs to be constructed from the AIG, which in the worst-case is exponential, since constructing the BDD that computes an AND node may require the product of the nodes used in the BDDs for each child function. The algorithm to construct a Pseudo-Kronecker expression from the BDD is cubic in the number of BDD nodes, since for each pair of co-factors one needs to compute the BDD that computes the XOR of both co-factors.

2) ESOP Minimization: In our implementation we use *exorcism* [32] to minimize the number of terms in the ESOP expression. Exorcism is a heuristic minimization algorithm that applies local rewriting of product terms using the *EXORLINK* operation [49]. In order to introduce this operation, we need to define the notation of distance. For each product term a variable can either appear as positive literal, as negative literal, or not at all. The *distance* of two product terms is the number of variables with different appearance in each term. For example, the two product terms x_1x_3 and $x_1\bar{x}_2x_3$ have distance 1, since x_2 does not appear in the first product term and appears as negative literal in the second one. It can be shown that two product terms with distance k can be rewritten as an equivalent ESOP expression with k product terms in $k!$ different ways. The EXORLINK-k operation

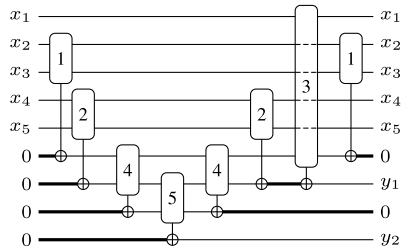

Fig. 7. Reversible network from Fig. 4(d). Additional lines which have a constant 0 value are drawn thicker. These lines can be used as additional resources when mapping single-target gates.

is a procedure to enumerate all $k!$ replacements for a product term pair with distance k. Applying the EXORLINK operation to product term pairs with a distance of less than 2 immediately leads to a reduction of the number of product terms in an ESOP expression. In fact, as described above, such checks are already applied when creating the initial cover. Applying the EXORLINK-2 operation does not increase the number of product terms but can decrease it, if product terms in the replacement can be combined with other terms. The same applies for distances larger than 2, but it can also lead to an increase in the number of product terms. This can sometimes be helpful to escape local minima. Exorcism implements a default minimization heuristic, referred to as *def* in the following, that applies different combinations and sequences of EXORLINK-k operations for $2 \leq k \leq 4$. We have modified the heuristic by just omitting the EXORLINK-4 operations, referred to as *def_wo4* in the following. The complexity of the exorcism algorithm is cubic in the number of initial product terms, since it checks whether product terms resulting from an EXORLINK operation to a pair of product terms can optimize any existing product term.

B. LUT-Based Mapping

This section describes a mapping technique that exploits two observations: 1) when mapping a single-target gate there may be additional lines available with a constant 0 value and 2) for single-target gates with few control lines (e.g., up to 4) one can precompute near-optimal Clifford+T circuits and store them in a database. Fig. 6(b) illustrates the LUT-based mapping flow. The idea is to apply 4-LUT mapping on the control function of the single-target gate and use available 0-valued ancilla lines to store intermediate results from inner LUTs in the mapping. If enough 0-valued ancilla lines are available, each of the single-target gates resulting from this mapping is directly translated into a near-optimum Clifford+T network that is looked up in a database. The size of the database is compressed by making use of Boolean function classification based on affine input transformations and output complementation. This procedure requires no additional lines being added to the circuit. If the procedure cannot be applied due to too few 0-valued ancilla lines, direct mapping is used as fallback.

1) Available 0-Valued Ancilla Resources: Fig. 7 shows the same reversible network as in Fig. 4(d) that is obtained from the initial k-LUT mapping. However, additional lines that are 0-valued are drawn thicker. We can see that for the realization of the first single-target gate, there are three 0-valued lines, but for the last single-target gate there is only one 0-valued line. This information can easily be obtained from the reversible network resulting from Algorithm 2.

The following holds in general. Let $g = T_c(X, t)$ be a single-target gate with $|X| = k$ and let there be l available 0-valued

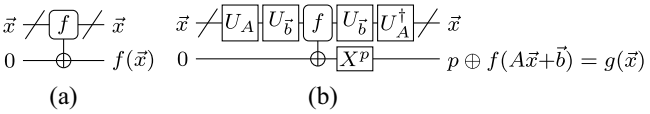

Fig. 8. Single-target gate for (a) T_f and (b) T_g, which can be constructed from T_f, since f and g are AN-equivalent. (a) T_f. (b) T_g.

lines, besides a 0-valued line for target line t. If c can be realized as a 4-LUT network with at most $l + 1$ LUTs, then we can realize it as a reversible network with $2l - 1$ single-target gates that realizes function c on target line t such that all inner LUTs compute and uncompute their results on the l available 0-valued lines. This synthesis procedure is similar but much simpler than Algorithm 2, since c is a single-output function. Therefore, the number of additional lines required for the inner LUTs cannot be improved by a different topological order and is solely determined by the number of LUTs in the mapping. Given the k-LUT mapping for the control function, the synthesis procedure is linear in the number of LUTs.

2) Near-Optimal Four-Input Single-Target Gates: There exists 2^{2^n} Boolean functions over n variables, i.e., 4, 16, 256, and 65 536 for $n = 1, 2, 3$, and 4, respectively. Consequently, there are as many different single-target gates $T_c(\{x_1, \ldots, x_n\}, x_{n+1})$. For this number, it is possible to precompute optimum or near-optimal Clifford+T circuits for each of the single-target gates using exact or heuristic optimization methods for Clifford+T gates (see [6], [50]–[52]), and store them in a database. Mapping single-target gates resulting from the LUT-based mapping technique described in this sections is therefore very efficient. However, the number of functions can be reduced significantly when using *affine function classification*. Next, we review affine function classification and show that two optimum Clifford+T circuits for two single-target gates with affine equivalent functions have the same T-count.

For a Boolean function $f(x_1, \ldots, x_n)$, let us use the notation $f(\vec{x})$, where $\vec{x} = (x_1, \ldots, x_n)^T$. We say that two functions f and g are *affine equivalent* [53], if there exists an $n \times n$ invertible matrix $A \in \mathrm{GL}_n(\mathbb{B})$ and a vector $\vec{b} \in \mathbb{B}^n$ such that

$$g(\vec{x}) = f\left(A\vec{x} + \vec{b}\right) \qquad \text{for all } \vec{x} \in \mathbb{B}^n. \tag{4}$$

We say that f and g are *affine equivalent under negation* [54], if either (4) holds or $g(\vec{x}) = \bar{f}(A\vec{x} + \vec{b})$ for all $\vec{x}$. For the sake of brevity, we say that f is *AN-equivalent* to g in the remainder. AN-equivalence can be considered an extension of NPN-equivalence, where besides input negation, input permutation, and output negation, also inputs x_i may be replaced by $x_i \oplus x_j$, where $j \neq i$. AN-equivalence is an equivalence relation and allows to partition the set of 2^{2^n} into much smaller sets of functions. For $n = 1, 2, 3$, and 4, there only 2, 3, 6, and 18 classes of AN-equivalent functions (see [53]–[55]). And all 4 294 967 296 5-input Boolean functions fall into only 206 classes of AN-equivalent functions! The database solution proposed in this mapping can therefore scale for five variables given a fast way to classify functions. Before we discuss classification algorithms, the following lemma shows that AN-equivalent functions preserve T-cost.

Lemma 3: Let f and g be two n-variable AN-equivalent functions. Then the T-count in Clifford+T circuits realizing T_f and T_g is the same.

Proof: Since f and g are AN-equivalent, there exists $A \in \mathrm{GL}_n(\mathbb{B})$, $\vec{b} \in \mathbb{B}^n$, and $p \in \mathbb{B}$ such that $g(\vec{x}) = p \oplus f(A\vec{x} + \vec{b})$ for all $\vec{x}$. It is possible to create a reversible circuit that takes

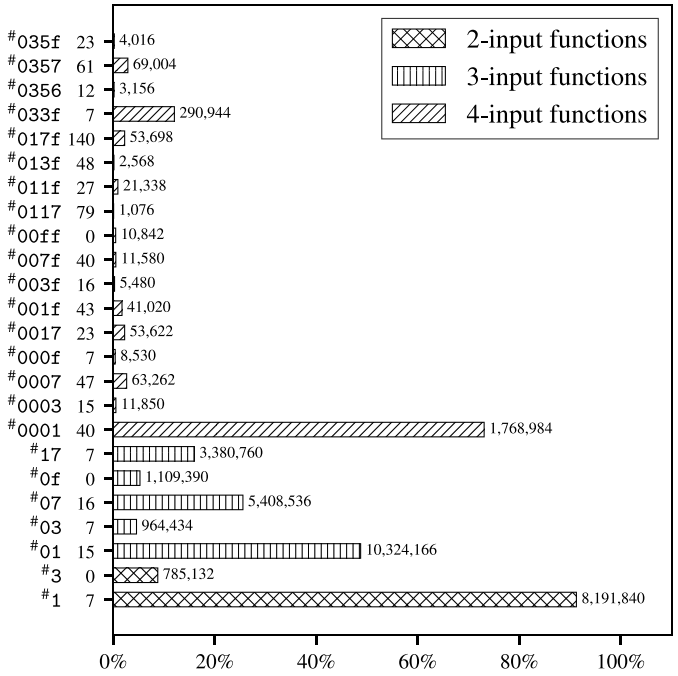

Fig. 9. This plot shows the distribution of AN-equivalence classes among all four-LUTs that have been discovered in all our experimental results (see Section VI). The y-axis shows from the bottom to the top all 2, 5, and 17 nonconstant AN-equivalence classes of 2, 3, and 4 variables. The number right to the truth table is the T-count in the best known Clifford+T realization of the corresponding single-target gate. The x-axis shows the frequency percentage with respect to other classes that have the same number of variables. The small number next to the bar shows the frequency in absolute values.

$\vec{x} \mapsto A\vec{x} + \vec{b}$ using only CNOT gates for A and NOT gates for $\vec{b}$ (see [56]). The output can be inverted using a NOT gate. Fig. 8 illustrates the proof. The subcircuit U_A realizes the linear transformation using CNOT gates, $U_{\vec{b}}$ realizes the input inversions using NOT gates, and X^p represents a NOT gate, if $p = 1$, otherwise the identity. ■

In order to make use of the algorithm we need to compute an optimum or near-optimal circuit for one representative in each equivalence class for up to four variables. To match an arbitrary single-target gate with a control function of up to four variables in the database, one needs to derive its representative. Algorithms as presented in [57] can be used for this purpose. They need to explore all $2^n \prod_{i=0}^{n-1}(2^n - 2^i)$ possible affine transformations as in (4) to the control function and its complement. Since $n \leq 4$, this enumeration is feasible.

Fig. 9 lists all the AN-equivalent classes for 2–4 variables; the class containing the constant functions is omitted. It shows how often they are discovered in a four-LUT in all our experimental evaluations. Also, it shows the number of T gates in the currently best-known Clifford+T circuits of the corresponding single-target gate found using the optimization heuristic in [50]. Classes #1, #01, #07, #17, and #0001 occur most frequently. The classes #1, #01, and #0001 contain among others x_1x_2, $x_1x_2x_3$, and $x_1x_2x_3x_4$, respectively, and are therefore control functions of the multiple-controlled Toffoli gates. Class #17 contains the majority-of-three function. The table can guide to the classes which benefit most from optimizing their T-count. Reversible and Clifford+T circuits of the currently best-known realizations can be found at quantumlib.stationq.com.

C. Hybrid Mapping

LUT-based mapping cannot be applied if the number of available 0-valued lines is insufficient. As fallback, the four-LUT network is omitted and direct mapping is applied on the k-LUT network, therefore not using any of the 0-valued lines at all. The idea of hybrid synthesis is to merge four-LUTs into larger LUTs. By merging two LUTs in the network, the number of LUTs is decreased by 1 and therefore one fewer 0-valued line is required. Our algorithm for hybrid synthesis merges the output LUT with one of its children, thereby generating a larger output LUT. This procedure is repeated until the LUT network is small enough to match the number of 0-valued lines. The topmost LUT is then mapped using direct mapping, while the remaining LUTs are translated using the LUT-mapping technique. If the number of leaves in the topmost LUT exceed the number of inputs of the control function, direct mapping is applied to the control function.

VI. Experimental Evaluation

We have implemented LHRS as command "lhrs" in the open source reversible logic synthesis framework RevKit [62].[1] All experiments have been carried out on an Intel Xeon CPU E5-2680 v3 at 2.50 GHz with 64 GB of main memory running Linux 4.4 and gcc 5.4.

A. LHRS Configuration

Table I gives an overview of all parameters that can be given as input to LHRS. The parameters are split into two groups. Parameters in the upper half have mainly an influence on the number of lines and are used to synthesize the initial reversible network that is composed of single-target gates (Section IV). Parameters in the lower half only influence the number of T gates by changing how single-target gates are mapped into Clifford+T circuits. Each parameter is shown with possible values and some explanation text.

B. Evaluating the Effects of Parameters

As benchmarks, we used the ten arithmetic instances of the EPFL combinational logic synthesis benchmarks [64], which are publicly available and commonly used to evaluate logic synthesis algorithms. In order to investigate the influence of the initial logic representation, we used different realizations of the benchmarks: 1) the original benchmark description in terms of an AIG, called *Original* and 2) the best known 6-LUT network wrt. the number of lines, called *Best-LUT*.[2] Further statistics about the benchmarks are given in Table II. All experimental results and synthesis outcomes for the EPFL arithmetic benchmarks and for a variety of IEEE-compliant floating point benchmarks can be viewed and downloaded from a repository github.com/msoeken/lhrs-experiments. The repository also contains all scripts to generate the results, making it easy to perform similar experiments on a different set of benchmarks.

We evaluated LHRS for both realizations (*Original* and *Best-LUT*) for all ten arithmetic benchmarks with an LUT size of 6, 10, and 16. Table III lists all results. We chose LUT size 6, because it is a typical choice for FPGA mapping, and therefore we expect that LUT mapping algorithms perform well for this size. We chose 16, since we noticed in our

[1]The source code can be found at github.com/msoeken/cirkit.

[2]See lsi.epfl.ch/benchmarks, version 2017.1.

TABLE I
PARAMETERS FOR LHRS

Parameter	Values	Description
Parameters p_{Q} that affect the number of qubits (and T gates)		
LUT size	$\{3, \ldots, 32\}$	Maximum number of inputs to LUTs in the LUT mapping, default: 16
Topo. order	*topo_def*, *topo_tfi_sort*	Heuristic in which order to traverse LUTs in the LUT mapping (Sect. IV-C), default: *topo_def*
area iters. (init)	$\{1, \ldots, 10\}$	Number of iterations for global area recovery heuristic based on exact area [25], [58], default: 2
flow iters. (init)	$\{1, \ldots, 10\}$	Number of iterations for local area recovery heursitic based on area flow [59], [60], default: 1
Parameters p_{T} that only affect the number of T gates		
Mapping	*direct*, *hybrid*	Mapping method (Sect. V), default: *hybrid*
ESOP extraction	*AIG extract*, *BDD extract*	ESOP extraction method (Sect. V-A), default: *BDD extract*
ESOP heuristic	*none*, *def*, *def_wo4*	ESOP minimization heuristic (Sect. V-A), *none* means that no optimization is applied, default: *def_wo4*
Parameters in *hybrid* mapping method		
area iters.	$\{1, \ldots, 10\}$	*see above*
flow iters.	$\{1, \ldots, 10\}$	*see above*
SAT-based opt.	*true*, *false*	Enables whether 4-LUT networks in the hybrid mapping are post-optimized using the SAT-based technique described in [61], default: *false*

TABLE II
EPFL ARITHMETIC BENCHMARKS

Benchmark	Inputs	Outputs	Original		Best-LUT	
			AIG nodes	Levels	LUTs	Levels
Adder	256	129	1020	255	192	64
Barrel shifter	135	128	3,336	12	512	4
Divisor	128	128	44,762	4,470	3268	1,208
Hypotenuse	256	128	214,335	24,801	40,406	4,532
Log2	32	32	32,060	444	6,574	119
Max	512	130	2,865	287	523	189
Multiplier	128	128	27,062	274	4,923	90
Sine	24	25	5,416	225	1,229	55
Square-root	128	64	24,618	5,058	3,077	1,106
Square	64	128	18,484	250	3,246	74

experiments that it is the largest LUT size for which LHRS performs reasonably fast for most of the benchmarks. LUT size 10 has been chosen, since it is roughly in between the other two. These configurations allow us to synthesize six different initial single-target gate networks for each benchmark, therefore spanning six optimization points in a Pareto set. For each of these configurations, we chose four different configurations of parameters in p_T, based on values to the mapping method and the ESOP heuristic ($\{$*direct*, *hybrid*$\} \times \{$*def*, *def_wo4*$\}$).

The experiments confirm the observation of Section IV-D. A larger LUT size leads to a smaller number qubits. In some cases, e.g., Log2, this can be quite significant. A larger LUT size also leads to a larger T-count, again very considerable, e.g., for Log2.

Slightly changing the ESOP minimization heuristic (note that *def* to *def_wo4* are very similar) has no strong influence on the number of T gates. There are examples for both the case in which the T-count increases and in which it decreases. However, the runtime can be significantly reduced. The choice of the mapping method has a stronger influence. For example, in case of the *Adder*, the *hybrid* mapping method is far superior compared to the *direct* method. In many cases, the *hybrid* method is advantageous only for an LUT size of 16, but not for the smaller ones. Also, the initial representation of the benchmark has a considerable influence. In many cases, the *Best-LUT* realizations of the benchmarks require fewer qubits, which—as expected—results in higher T-count. The effect is particularly noticeable for the *Divisor* and *Square-root*. In some cases, better results in both qubits and T-count can be achieved, e.g., for Log2 as *Best-LUT* with an LUT size of 16, and for the *Barrel shifter* as *Original* with an LUT size of 16.

C. Comparison to State-of-the-Art Algorithms

We compare LHRS to three state-of-the-art hierarchical synthesis algorithms: circuit-based synthesis (CBS, [9]), XOR-majority graph (XMG)-based synthesis (DXS, [10]), and BDD-based synthesis [63]. Before we discuss the results from the experimental evaluation, we compare the three state-of-the-art approaches conceptually to LHRS.

1) Comparison to CBS: CBS uses a logic network as starting point, in its current implementation an AIG. The scalability of the approach therefore depends on the size of the logic network, which makes it comparable to LHRS. CBS decomposes the logic network into multioutput subnetworks based on adjacent and overlapping fan-out free regions. These are embedded and synthesized using explicit or symbolic functional reversible logic synthesis algorithms (depending on the size of the subnetworks). The size of the subnetworks can be controlled with a threshold parameter t. A larger value for t usually leads to a smaller number of qubits, but if t is chosen to be too large, the embedding and synthesis algorithms may take a very long time.

In LHRS, we decompose the initial logic network into k-input (single-output) LUTs, which are represented as single-target gates in the intermediate reversible logic network. We use a variety of different synthesis algorithms to map these single-target gates into quantum circuits. These approaches are more scalable as compared to the functional reversible logic synthesis algorithms used in CBS. The most time-consuming mapping algorithm that we propose and use in the implementation of LHRS is *direct mapping* (Section V-A). It is based on ESOP-based synthesis, which was demonstrated to scale better (in runtime) to functional reversible logic synthesis approaches (see [10, Tables II and III]).

The computational complexity of both approaches is the same, and exponential in the worst-case. In CBS with threshold value t, it can be that almost all 2^{2t} input/output patterns need to be visited by the functional reversible logic synthesis algorithm. In LHRS with LUT size k, it can be that ESOP-based synthesis generates reversible circuits with $\approx 2^k$ Toffoli gates. Note that both methods are polynomial in the number of logic gates of the initial logic representation, since both k and t are constant and fixed parameters that are input to the synthesis problem.

2) Comparison to DXS: DXS uses an XMG as a starting point, and is therefore as scalable as CBS and LHRS, since it depends on the size of the initial logic network. LHRS has therefore no scalability advantage over DXS. Also, DXS uses ancillae management strategies very similar to LHRS. DXS is best considered a special case of LHRS in which $k = 3$ and only two LUT functions, namely XOR and majority-of-three

TABLE III
EXPERIMENTAL EVALUATION OF LHRS ON THE EPFL ARITHMETIC BENCHMARKS

Benchmark	LUT size		*def*, *direct*			*def_wo4*, *direct*		*def*, *hybrid*		*def_wo4*, *hybrid*	
			qubits	T-count	runtime [s]	T-count	runtime [s]	T-count	runtime [s]	T-count	runtime [s]
Adder	6	Best-LUT	448	20,487	1.00	14,411	0.99	21,121	0.37	21,023	0.43
		Original	505	6,750	0.99	6,260	1.03	1,988	0.04	1,988	0.05
	10	Best-LUT	445	24,320	1.11	18,558	1.20	22,058	0.46	21,953	0.46
		Original	490	20,889	1.34	20,463	1.40	2,730	0.08	2,730	0.05
	16	Best-LUT	443	50,490	2.35	44,628	1.49	23,378	0.54	23,273	0.64
		Original	463	295,130	11.82	293,737	7.33	4,914	0.05	4,914	0.10
Barrel shifter	6	Best-LUT	840	32,512	4.85	32,512	4.86	17,024	0.08	17,024	0.05
		Original	584	50,944	3.36	50,944	2.49	108,917	0.66	108,917	0.95
	10	Best-LUT	740	52,986	4.65	52,986	4.72	25,676	0.16	25,676	0.08
		Original	584	50,944	3.52	50,944	3.48	108,917	0.70	108,917	0.96
	16	Best-LUT	595	114,690	3.04	114,690	4.59	47,390	0.10	47,390	0.25
		Original	582	52,480	2.56	52,480	2.56	110,669	0.56	110,669	0.82
Divisor	6	Best-LUT	3,399	344,060	23.44	371,193	23.29	405,855	3.44	405,896	3.31
		Original	12,389	754,601	351.81	754,609	356.18	729,940	7.78	729,940	11.40
	10	Best-LUT	3,226	448,622	21.58	473,355	18.82	466,594	2.74	466,635	2.67
		Original	12,055	877,437	248.58	874,085	306.71	874,476	6.71	874,476	6.72
	16	Best-LUT	3,017	3,451,565	658.10	3,497,302	342.84	882,852	2.99	882,893	2.80
		Original	11,827	1,403,597	267.81	1,409,227	367.55	1,044,271	7.22	1,044,271	6.94
Hypotenuse	6	Best-LUT	40,611	3,762,930	601.54	3,819,132	587.01	6,495,809	249.58	6,495,823	371.59
		Original	47,814	2,357,672	616.85	2,316,690	622.27	3,021,920	182.66	3,021,920	174.50
	10	Best-LUT	36,444	5,861,897	566.04	5,966,706	371.11	8,789,723	232.55	8,789,664	250.10
		Original	43,871	4,337,100	604.58	4,334,308	611.90	4,330,903	323.69	4,330,883	217.42
	16	Best-LUT	32,309	16,339,405	2,500.12	16,553,536	875.15	11,939,791	246.44	11,939,196	366.14
		Original	39,324	16,747,558	3,829.30	16,864,051	1,297.15	6,195,013	339.82	6,194,774	325.07
Log2	6	Best-LUT	6,627	659,917	81.19	661,101	92.42	860,028	20.96	860,028	20.35
		Original	7,611	500,919	88.62	500,961	75.84	628,777	5.17	628,777	5.04
	10	Best-LUT	3,038	1,974,254	100.08	1,984,321	108.49	14,116,511	85.62	14,116,708	74.62
		Original	2,875	3,205,679	111.67	3,220,755	98.10	15,188,601	88.69	15,188,782	91.37
	16	Best-LUT	2,054	41,820,984	22,994.62	42,712,038	7,662.99	33,707,671	1,283.19	33,794,883	648.48
		Original	2,315	48,342,197	29,463.76	49,489,819	9,403.78	41,174,178	906.50	41,207,475	587.85
Max	6	Best-LUT	1,036	62,901	7.70	56,911	6.00	18,369	0.66	18,369	1.30
		Original	1,233	74,090	7.37	68,664	8.46	48,490	2.27	48,490	1.60
	10	Best-LUT	840	193,048	4.31	187,482	3.52	23,460	0.25	23,460	0.36
		Original	901	222,063	8.63	217,721	3.84	55,993	0.72	55,993	0.63
	16	Best-LUT	725	1,017,728	32.87	1,014,184	52.92	32,388	0.26	32,380	0.49
		Original	796	732,971	20.15	729,111	8.66	64,089	1.29	63,993	1.31
Multiplier	6	Best-LUT	5,048	682,141	56.32	684,723	53.73	929,136	8.95	929,110	5.81
		Original	5,806	386,900	99.57	387,236	98.28	399,632	3.14	399,632	7.87
	10	Best-LUT	3,418	1,221,457	47.64	1,225,308	42.71	1,288,887	5.65	1,288,912	10.40
		Original	3,105	2,743,654	56.17	2,745,190	40.22	1,035,456	3.74	1,035,633	10.48
	16	Best-LUT	2,552	9,246,977	1,030.50	9,337,179	587.85	2,580,526	45.70	2,581,486	78.11
		Original	2,852	8,070,214	905.87	8,147,508	636.44	1,298,702	16.34	1,299,740	11.11
Sine	6	Best-LUT	1,277	139,874	9.45	140,232	7.30	195,187	7.51	195,203	7.84
		Original	1,468	88,008	25.10	87,494	8.79	109,248	5.20	109,248	5.25
	10	Best-LUT	558	351,993	6.18	352,360	4.59	603,919	14.26	603,866	26.14
		Original	714	412,680	8.78	414,156	8.56	660,934	6.55	661,028	6.51
	16	Best-LUT	415	2,226,719	356.40	2,262,039	167.57	1,567,912	157.05	1,581,160	93.50
		Original	518	3,830,542	1,076.41	3,883,236	437.53	1,937,829	12.73	1,938,452	11.96
Square-root	6	Best-LUT	3,204	364,314	14.23	383,145	47.47	407,040	3.26	407,081	6.54
		Original	8,212	279,303	29.68	279,237	44.00	712,757	2.04	712,757	2.00
	10	Best-LUT	2,874	541,601	13.71	559,224	12.22	508,059	4.01	508,105	3.67
		Original	7,892	323,522	30.64	323,756	30.00	758,738	4.57	758,762	1.58
	16	Best-LUT	2,628	6,162,475	326.39	6,169,246	255.41	1,250,236	28.63	1,251,139	28.74
		Original	7,816	1,206,310	143.00	1,213,454	151.56	861,535	4.50	861,567	2.04
Square	6	Best-LUT	3,309	301,646	36.38	297,268	39.42	651,639	7.75	651,639	5.53
		Original	4,058	195,306	17.51	195,306	17.41	504,369	3.40	504,369	3.58
	10	Best-LUT	2,882	531,820	13.09	530,984	31.96	911,933	7.04	911,949	8.14
		Original	3,355	463,373	36.66	463,825	45.67	781,317	4.46	781,512	4.39
	16	Best-LUT	2,298	2,747,513	1,686.09	2,788,657	361.14	1,933,138	800.26	1,950,409	232.57
		Original	2,664	2,832,786	1,522.35	2,882,096	392.68	1,873,406	898.58	1,898,077	219.48

(incl. variants with input complementation), are possible. For these, compact quantum circuit realizations are known.

As our experimental results will demonstrate, DXS can be seen as a complementary method to LHRS for small LUT sizes. DXS scales better than LHRS with $k = 3$, because 1) only functions with inexpensive quantum circuit realizations are used in DXS and 2) dedicated optimization routines to optimize XMGs can be exploited. But due to the small gate sizes, DXS does not easily reduce the number of required qubits.

In Section V-B, we discuss the use of function classification (AN-classification), since each function in an equivalence class has the same number of T gates in an T-count optimum quantum circuit. The statistical information shown in Fig. 9 motivates to investigate dedicated LUT mapping algorithms that favor LUT functions with smaller T-count. In this way, LHRS can achieve a similar quality compared to specialized approaches, such as DXS.

3) Comparison to BDD-Based Synthesis: In BDD-based synthesis, first the input function is represented as binrary

TABLE IV
COMPARISON TO STATE-OF-THE-ART ALGORITHMS (QUALITY OF RESULTS)

Benchmark		CBS [9]		DXS [10]		BDD-based [63]		LHRS $k = 6$		LHRS $k = 10$		LHRS $k = 16$	
		qubits	T gates	qubits	T gates	qubits	T gates	qubits	T gates	qubits	T gates	qubits	T gates
Adder	Best-LUT	1,023	1,973,318	701	6,692			448	21,023	445	21,953	443	23,273
	Original	640	427,896	386	1,785			505	1,988	490	2,730	463	4,914
Barrel shifter	Best-LUT	748	616,754	1,737	11,648			840	17,024	740	25,676	595	47,390
	Original	935	1,646,314	1,359	36,036			584	108,917	584	108,917	582	110,669
Divisor	Best-LUT	12,215	58,064,206	15,411	158,774			3,399	405,896	3,226	466,635	3,017	882,893
	Original	28,567	36,344,794	21,166	468,489			12,389	729,940	12,055	874,476	11,827	1,044,271
Hypotenuse	Best-LUT	141,505	661,691,082	141,410	1,363,614			40,611	6,495,823	36,444	8,789,664	32,309	11,939,196
	Original	91,272	152,597,174	125,433	1,113,392			47,814	3,021,920	43,871	4,330,883	39,324	6,194,774
Log2	Best-LUT	15,739	55,549,274	29,461	357,553			6,627	860,028	3,038	14,116,708	2,054	33,794,883
	Original	8,811	22,029,122	21,625	245,868			7,611	628,777	2,875	15,188,782	2,315	41,207,475
Max	Best-LUT	994	2,846,164	1,611	10,920			1,036	18,369	840	23,460	725	32,380
	Original	1,157	2,873,590	2,493	23,268			1,233	48,490	901	55,993	796	63,993
Multiplier	Best-LUT	13,105	44,389,136	18,967	223,398			5,048	929,110	3,418	1,288,912	2,552	2,581,486
	Original	5,548	6,771,882	15,691	159,271			5,806	399,632	3,105	1,035,633	2,852	1,299,740
Sine	Best-LUT	2,929	10,503,022	5,807	72,037	890,414	37,920,330	1,277	195,203	558	603,866	415	1,581,160
	Original	1,714	4,041,028	3,818	43,778	890,414	37,920,330	1,468	109,248	714	661,028	518	1,938,452
Square-root	Best-LUT	12,000	65,083,062	15,672	160,566			3,204	407,081	2,874	508,105	2,628	1,251,139
	Original	6,778	28,627,150	17,485	215,068			8,212	712,757	7,892	758,762	7,816	861,567
Square	Best-LUT	12,348	44,222,254	10,584	118,272			3,309	651,639	2,882	911,949	2,298	1,950,409
	Original	6,479	9,837,268	12,091	130,445			4,058	504,369	3,355	781,512	2,664	1,898,077
Geomean		1.75	56.64	2.60	0.38			1.0	1.0	0.75	1.95	0.64	3.34

decision diagram. Then each node is translated into a corresponding quantum circuit similar to DXS, where XOR and majority gates are translated. However, BDD-based synthesis relies on an efficient construction of a BDD for a given input function. If a BDD can be obtained within reasonable time and particularly within the available memory resources, the synthesis procedure is very fast. However, it is known that the size of a BDD is exponential in the worst-case and also in the average case. Large functions often suffer from the memory explosion problem. Also, the canonicity property of BDDs prohibits logic optimization besides variable reordering.

Techniques, which are discussed in our proposed decomposition approach, may also be applied to the BDD-based approach. The ancillae management heuristic described in Section IV and Algorithm 2 could in a similar way be applied to BDD-based synthesis in order to uncompute ancillae early and reuse them in order to reduce the number of overall required ancillae. However, this still requires that the BDD must be constructed and does not address the scalability issue.

In our experimental evaluation we set $t = 10$ for CBS, which results in a similar number of additional lines compared to LHRS with LUT size $k = 6$. It is important to note that CBS and BDD-based synthesis do not uncompute results and produce garbage outputs. For a fair comparison, we update the numbers returned by the synthesis algorithm, and use the Bennett trick [14] to uncompute all garbage lines. This trick requires to add one ancilla for each output and to double the number of T gates. We have set a memory limit of 100 GB. This affects particularly BDD-based synthesis, which requires high memory resources for the construction of the BDD. For LHRS, we compare LUT sizes 6, 10, and 16, mapping method *hybrid*, and ESOP heuristic *def_wo4*.

Table IV list the quality of results for the comparison with the state-of-the-art techniques. For each approach we list the number of qubits and the number of T gates. The runtimes are listed in Table V. The last row in Table IV lists the geomean for both qubits and T gates normalized to the results of LHRS for $k = 6$. The results demonstrate that LHRS clearly outperforms CBS in terms of T-count, while often having a similar number of qubits. For no benchmark, CBS produced a Pareto-optimal result. LHRS outperforms DXS in terms of qubits, except for the *Adder* benchmark. DXS always produces at least one Pareto-optimal result. With the memory limit of 100 GB, it was only possible to apply the BDD-based approach to the *Sine* benchmark. But for that function, the resulting qubit numbers and T-count are several magnitudes higher than those obtained by LHRS. Since we only have results for a single benchmark, we omit the geomean for BDD-based synthesis. LHRS produces Pareto-optimal results for all benchmarks except *Adder*.

TABLE V
COMPARISON TO STATE-OF-THE-ART ALGORITHMS (RUNTIME)

Benchmark		CBS	DXS	BDD-based	$k = 6$	$k = 10$	$k = 16$
Adder	B	43.77	0.04	MO	0.43	0.46	0.64
	O	5.01	0.01	MO	0.05	0.05	0.10
Barrel shifter	B	10.36	0.11	MO	0.05	0.08	0.25
	O	36.14	0.10	MO	0.95	0.96	0.82
Divisor	B	2,461.32	14.16	MO	3.31	2.67	2.80
	O	735.28	19.43	MO	11.40	6.72	6.94
Hypotenuse	B	20,165.43	827.20	MO	371.59	250.10	366.14
	O	3,429.19	762.55	MO	174.50	217.42	325.07
Log2	B	1,789.86	28.29	MO	20.35	74.62	648.48
	O	471.89	13.69	MO	5.04	91.37	587.85
Max	B	46.07	0.18	MO	1.30	0.36	0.49
	O	89.65	0.38	MO	1.60	0.63	1.31
Multiplier	B	1,213.33	11.80	MO	5.81	10.40	78.11
	O	122.55	7.67	MO	7.87	10.48	11.11
Sine	B	316.16	0.99	3,207.84	7.84	26.14	93.50
	O	99.58	0.46	2,998.05	5.25	6.51	11.96
Square-root	B	2,584.43	7.18	MO	6.54	3.67	28.74
	O	412.80	9.82	MO	2.00	1.58	2.04
Square	B	1,226.92	3.59	MO	5.53	8.14	232.57
	O	207.44	4.64	MO	3.58	4.39	219.48

All runtimes are given in seconds. Entries 'MO' indicate that synthesis was aborted after the memory limit of 100 GB has been reached.

VII. CONCLUSION

We presented LHRS, an LUT-based approach to hierarchical reversible circuit synthesis that outperforms existing state-of-the-art hierarchical methods. Its two-stage approach allows us to first find an abstract single-target gate reversible logic network that already determines the number of required qubits. Several different configuration parameters and mapping strategies allow for a fast exploration of single-target gate networks with the aim to find one with a suitable number of

qubits. In the second step, each single-target gate is mapped into a quantum circuit, in this case using Clifford+T gates.

LHRS can be regarded as a synthesis framework since it consists of several parts that can be optimized separately. As one example, we are currently investigating more advanced mapping strategies that map single-target gates into Clifford+T circuits. Also, most of the conventional synthesis approaches that are used in the LHRS flow, e.g., the mapping algorithms to derive the k-LUT network, are not quantum-aware, i.e., they do not explicitly optimize wrt. the quality of the resulting quantum network. We expect further improvements, particularly in the number of T gates, by modifying the synthesis algorithms in that direction. Finally, to address the tight qubit resources, we plan to employ reversible pebbling strategies to find initial single-target gate networks with fewer qubits, for the expense of a higher gate count.

Acknowledgment

The authors would like to thank M. Amy, O. Di Matteo, V. Kliuchnikov, G. Meuli, and A. Mishchenko for many useful discussions. All circuits in this paper were drawn with the *qpic* tool.

References

[1] M. Soeken, M. Roetteler, N. Wiebe, and G. De Micheli, "Hierarchical reversible logic synthesis using LUTs," in *Proc. Design Autom. Conf.*, Austin, TX, USA, 2017, p. 78.

[2] S. Debnath *et al.*, "Demonstration of a small programmable quantum computer with atomic qubits," *Nature*, vol. 536, pp. 63–66, Aug. 2016.

[3] N. M. Linke *et al.*, "Experimental comparison of two quantum computing architectures," *Proc. Nat. Acad. Sci. USA*, vol. 114, no. 13, pp. 3305–3310, 2017.

[4] E. A. Martinez *et al.*, "Real-time dynamics of lattice gauge theories with a few-qubit quantum computer," *Nature*, vol. 534, pp. 516–519, Jun. 2016.

[5] P. J. J. O'Malley *et al.*, "Scalable quantum simulation of molecular energies," *Phys. Rev. X*, vol. 6, no. 3, 2016, Art. no. 031007.

[6] 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. Comput.-Aided Design Integr. Circuits Syst.*, vol. 32, no. 6, pp. 818–830, Jun. 2013, doi: 10.1109/TCAD.2013.2244643.

[7] D. Maslov, "Advantages of using relative-phase Toffoli gates with an application to multiple control Toffoli optimization," *Phys. Rev. A*, vol. 93, no. 2, 2016, Art. no. 022311.

[8] M. Rawski, "Application of functional decomposition in synthesis of reversible circuits," in *Proc. Int. Conf. Reversible Comput.*, Grenoble, France, 2015, pp. 285–290, doi: 10.1007/978-3-319-20860-2_20.

[9] M. Soeken and A. Chattopadhyay, "Unlocking efficiency and scalability of reversible logic synthesis using conventional logic synthesis," in *Proc. Design Autom. Conf.*, Austin, TX, USA, 2016, p. 149. [Online]. Available: http://doi.acm.org/10.1145/2897937.2898107

[10] M. Soeken, M. Roetteler, N. Wiebe, and G. De Micheli, "Design automation and design space exploration for quantum computers," in *Proc. Design Autom. Test Europe*, Lausanne, Switzerland, 2017, pp. 470–475.

[11] D. Große, R. Wille, G. W. Dueck, and R. Drechsler, "Exact multiple-control Toffoli network synthesis with SAT techniques," *IEEE Trans. Comput.-Aided Design Integr. Circuits Syst.*, vol. 28, no. 5, pp. 703–715, May 2009, doi: 10.1109/TCAD.2009.2017215.

[12] D. M. Miller, D. Maslov, and G. W. Dueck, "A transformation based algorithm for reversible logic synthesis," in *Proc. Design Autom. Conf.*, Anaheim, CA, USA, 2003, pp. 318–323. [Online]. Available: http://doi.acm.org/10.1145/775832.775915

[13] M. Soeken, R. Wille, and R. Drechsler, "Hierarchical synthesis of reversible circuits using positive and negative Davio decomposition," in *Proc. Int. Design Test Symp.*, 2010, pp. 143–148, doi: 10.1109/IDT.2010.5724427.

[14] C. H. Bennett, "Logical reversibility of computation," *IBM J. Res. Develop.*, vol. 17, no. 6, pp. 525–532, 1973.

[15] J. J. Law and J. E. Rice, "Line reduction in reversible circuits using KFDDs," in *Proc. Pac. Rim Conf. Commun. Comput. Signal Process.*, Victoria, BC, Canada, 2015, pp. 113–118, doi: 10.1109/PACRIM.2015.7334819.

[16] R. Wille, M. Soeken, and R. Drechsler, "Reducing the number of lines in reversible circuits," in *Proc. Design Autom. Conf.*, Anaheim, CA, USA, 2010, pp. 647–652. [Online]. Available: http://doi.acm.org/10.1145/1837274.1837439

[17] K. Fazel, M. A. Thornton, and J. E. Rice, "ESOP-based Toffoli gate cascade generation," in *Proc. Pac. Rim Conf. Commun. Comput. Signal Process.*, Victoria, BC, Canada, 2007, pp. 206–209.

[18] F. T. Chong, D. Franklin, and M. Martonosi, "Programming languages and compiler design for realistic quantum hardware," *Nature*, vol. 549, no. 7671, pp. 180–187, 2017.

[19] D. Wecker and K. M. Svore, "LIQ$Ui|\rangle$: A software design architecture and domain-specific language for quantum computing," *arXiv preprint arXiv:1402.4467*, 2014.

[20] D. S. Steiger, T. Haener, and M. Troyer, "ProjectQ: An open source software framework for quantum computing," *Quantum*, vol. 2, p. 49, 2018.

[21] R. K. Brayton and A. Mishchenko, "ABC: An academic industrial-strength verification tool," in *Proc. Comput.-Aided Verification*, Edinburgh, U.K., 2010, pp. 24–40, doi: 10.1007/978-3-642-14295-6_5.

[22] J. Cong and Y. Ding, "FlowMap: An optimal technology mapping algorithm for delay optimization in lookup-table based FPGA designs," *IEEE Trans. Comput.-Aided Design Integr. Circuits Syst.*, vol. 13, no. 1, pp. 1–12, Jan. 1994, doi: 10.1109/43.273754.

[23] D. Chen and J. Cong, "DAOmap: A depth-optimal area optimization mapping algorithm for FPGA designs," in *Proc. Int. Conf. Comput.-Aided Design*, San Jose, CA, USA, 2004, pp. 752–759, doi: 10.1109/ICCAD.2004.1382677.

[24] A. Mishchenko, S. Cho, S. Chatterjee, and R. K. Brayton, "Combinational and sequential mapping with priority cuts," in *Proc. Int. Conf. Comput.-Aided Design*, San Jose, CA, USA, 2007, pp. 354–361, doi: 10.1109/ICCAD.2007.4397290.

[25] A. Mishchenko, S. Chatterjee, and R. K. Brayton, "Improvements to technology mapping for LUT-based FPGAs," *IEEE Trans. Comput.-Aided Design Integr. Circuits Syst.*, vol. 26, no. 2, pp. 240–253, Feb. 2007, doi: 10.1109/TCAD.2006.887925.

[26] S. Ray *et al.*, "Mapping into LUT structures," in *Proc. Design Autom. Test Europe*, Dresden, Germany, 2012, pp. 1579–1584, doi: 10.1109/DATE.2012.6176724.

[27] A. Kuehlmann, V. Paruthi, F. Krohm, and M. K. Ganai, "Robust Boolean reasoning for equivalence checking and functional property verification," *IEEE Trans. Comput.-Aided Design Integr. Circuits Syst.*, vol. 21, no. 12, pp. 1377–1394, Dec. 2002, doi: 10.1109/TCAD.2002.804386.

[28] L. G. Amarù, P.-E. Gaillardon, and G. De Micheli, "Majority-inverter graph: A new paradigm for logic optimization," *IEEE Trans. Comput.-Aided Design Integr. Circuits Syst.*, vol. 35, no. 5, pp. 806–819, May 2016, doi: 10.1109/TCAD.2015.2488484.

[29] A. De Vos and Y. Van Rentergem, "Young subgroups for reversible computers," *Adv. Math. Commun.*, vol. 2, no. 2, pp. 183–200, 2008, doi: 10.3934/amc.2008.2.183.

[30] G. Bioul, M. Davio, and J.-P. Deschamps, "Minimization of ring-sum expansions of Boolean functions," *Philips Res. Rep.*, vol. 28, pp. 17–36, Jan. 1973.

[31] S. Stergiou, K. Daskalakis, and G. K. Papakonstantinou, "A fast and efficient heuristic ESOP minimization algorithm," in *Proc. ACM Great Lakes Symp. VLSI*, Boston, MA, USA, 2004, pp. 78–81. [Online]. Available: http://doi.acm.org/10.1145/988952.988971

[32] A. Mishchenko and M. A. Perkowski, "Fast heuristic minimization of exclusive-sums-of-products," in *Proc. Reed Muller Workshop*, 2001.

[33] M. Saeedi and I. L. Markov, "Synthesis and optimization of reversible circuits—A survey," *ACM Comput. Surveys*, vol. 45, no. 2, p. 21, 2013. [Online]. Available: http://doi.acm.org/10.1145/2431211.2431220

[34] M. A. Nielsen and I. L. Chuang, *Quantum Computation and Quantum Information*. Cambridge, U.K.: Cambridge Univ. Press, 2000.

[35] D. Gosset, V. Kliuchnikov, M. Mosca, and V. Russo, "An algorithm for the T count," *Quant. Inf. Comput.*, vol. 14, nos. 15–16, pp. 1261–1276, 2014.

[36] P. Selinger, "Quantum circuits of T depth one," *Phys. Rev. A*, vol. 87, Apr. 2013, Art. no. 042302.

[37] N. Abdessaied, M. Amy, M. Soeken, and R. Drechsler, "Technology mapping of reversible circuits to Clifford $+T$ quantum circuits," in *Proc. Int. Symp. Multiple Valued Logic*, 2016, pp. 150–155, doi: 10.1109/ISMVL.2016.33.

[38] A. Barenco *et al.*, "Elementary gates for quantum computation," *Phys. Rev. A*, vol. 52, no. 5, p. 3457, 1995.

[39] C. Jones, "Low-overhead constructions for the fault-tolerant Toffoli gate," *Phys. Rev. A*, vol. 87, no. 2, 2013, Art. no. 022328.

[40] C. H. Bennett, "Time/space trade-offs for reversible computation," *SIAM J. Comput.*, vol. 18, no. 4, pp. 766–776, 1989, doi: 10.1137/0218053.

[41] R. Královic, "Time and space complexity of reversible pebbling," in *Proc. Conf. Current Trends Theory Pract. Informat.*, 2001, pp. 292–303, doi: 10.1007/3-540-45627-9_26.
[42] S. M. Chan, "Just a pebble game," in *Proc. Conf. Comput. Complexity*, Stanford, CA, USA, 2013, pp. 133–143, doi: 10.1109/CCC.2013.22.
[43] B. Komarath, J. Sarma, and S. Sawlani, "Reversible pebble game on trees," in *Proc. Int. Conf. Comput. Comb.*, 2015, pp. 83–94, doi: 10.1007/978-3-319-21398-9_7.
[44] A. Parent, M. Roetteler, and K. M. Svore, "REVS: A tool for space-optimized reversible circuit synthesis," in *Proc. Reversible Comput.*, Kolkata, India, 2017, pp. 90–101, doi: 10.1007/978-3-319-59936-6_7.
[45] B. D. O. Schmitt, A. Mishchenko, V. N. Kravets, R. K. Brayton, and A. I. Reis, "Fast-extract with cube hashing," in *Proc. Asia South Pac. Design Autom. Conf.*, 2017, pp. 145–150.
[46] M. Davio, J.-P. Deschamps, and A. Thayse, *Discrete and Switching Functions*. New York, NY, USA: McGraw-Hill, 1978.
[47] T. Sasao, "AND-EXOR expressions and their optimization," in *Logic Synthesis and Optimization*, T. Sasao, Ed. Boston, MA, USA: Kluwer, 1993.
[48] R. Drechsler, "Preudo–Kronecker expressions for symmetric functions," *IEEE Trans. Comput.*, vol. 48, no. 9, pp. 987–990, Sep. 1999, doi: 10.1109/12.795226.
[49] N. Song, "Minimization of exclusive sum of product expressions for mutliple-valued input incompletely specified functions," M.S. thesis, Elect. Eng. Dept., Portland State Univ., Portland, OR, USA, 1992.
[50] M. Amy, D. Maslov, and M. Mosca, "Polynomial-time T-depth optimization of Clifford $+T$ circuits via matroid partitioning," *IEEE Trans. Comput.-Aided Design Integr. Circuits Syst.*, vol. 33, no. 10, pp. 1476–1489, Oct. 2014, doi: 10.1109/TCAD.2014.2341953.
[51] O. D. Matteo and M. Mosca, "Parallelizing quantum circuit synthesis," *Quant. Sci. Technol.*, vol. 1, no. 1, 2016, Art. no. 015003.
[52] D. M. Miller, M. Soeken, and R. Drechsler, "Mapping NCV circuits to optimized Clifford $+T$ circuits," in *Proc. Int. Conf. Reversible Comput.*, 2014, pp. 163–175, doi: 10.1007/978-3-319-08494-7_13.
[53] M. A. Harrison, "On the classification of Boolean functions by the general linear and affine groups," *J. Soc. Ind. Appl. Math.*, vol. 12, no. 2, pp. 285–299, 1964. [Online]. Available: http://www.jstor.org/stable/2946369
[54] M. A. Harrison, "The number of equivalence classes of Boolean functions under groups containing negation," *IEEE Trans. Electron. Comput.*, vol. EC-12, no. 5, pp. 559–561, Oct. 1963, doi: 10.1109/PGEC.1963.263656.
[55] Y. Zhang, G. Yang, W. N. N. Hung, and J. Zhang, "Computing affine equivalence classes of Boolean functions by group isomorphism," *IEEE Trans. Comput.*, vol. 65, no. 12, pp. 3606–3616, Dec. 2016.
[56] M. Soeken, N. Abdessaied, and G. De Micheli, "Enumeration of reversible functions and its application to circuit complexity," in *Proc. Int. Conf. Reversible Comput.*, 2016, pp. 255–270, doi: 10.1007/978-3-319-40578-0_19.
[57] M. Soeken, I. Kodrasi, and G. De Micheli, "Boolean function classification with δ-swaps," in *Proc. Reed Muller Workshop*, 2017.
[58] J. Cong and Y. Ding, "On area/depth trade-off in LUT-based FPGA technology mapping," *IEEE Trans. Very Large Scale Integr. (VLSI) Syst.*, vol. 2, no. 2, pp. 137–148, Jun. 1994, doi: 10.1109/92.285741.
[59] J. Cong, C. Wu, and Y. Ding, "Cut ranking and pruning: Enabling a general and efficient FPGA mapping solution," in *Proc. Int. Symp. Field Program. Gate Arrays*, 1999, pp. 29–35. [Online]. Available: http://doi.acm.org/10.1145/296399.296425
[60] V. Manohararajah, S. D. Brown, and Z. G. Vranesic, "Heuristics for area minimization in LUT-based FPGA technology mapping," in *Proc. Int. Workshop Logic Synth.*, 2004, pp. 2331–2340.
[61] B. Schmitt, A. Mishchenko, and R. K. Brayton, "SAT-based area recovery in technology mapping," in *Proc. Int. Workshop Logic Synth.*, 2017.
[62] M. Soeken, S. Frehse, R. Wille, and R. Drechsler, "RevKit: A toolkit for reversible circuit design," *Multiple Valued Logic Soft Comput.*, vol. 18, no. 1, pp. 55–65, 2012. [Online]. Available: http://www.oldcitypublishing.com/MVLSC/MVLSCabstracts/MVLSC18.1abstracts/MVLSCv18n1p55-65Soeken.html
[63] R. Wille and R. Drechsler, "BDD-based synthesis of reversible logic for large functions," in *Proc. Design Autom. Conf.*, 2009, pp. 270–275. [Online]. Available: http://doi.acm.org/10.1145/1629911.1629984
[64] L. G. Amarù, P.-E. Gaillardon, and G. De Micheli, "The EPFL combinational benchmark suite," in *Proc. Int. Workshop Logic Synth.*, 2015.

Mathias Soeken (S'09–M'13) received the Ph.D. degree in computer science and engineering from the University of Bremen, Bremen, Germany, in 2013.

He is a Scientist with the École Polytechnique Fédéderale de Lausanne, Lausanne, Switzerland. His current research interests include logic synthesis and formal verification. He is investigating constraint-based techniques in logic synthesis and industrial-strength design automation for quantum computing. He is actively maintaining the logic synthesis frameworks CirKit and RevKit. He received a scholarship from the German Academic Scholarship Foundation.

Dr. Soeken has been serving as a TPC member for several conferences, including DAC, DATE, and ICCAD and is a Reviewer for *Mathematical Reviews* as well as for several journals.

Martin Roetteler (M'03) received the Ph.D. degree in computer science from the University of Karlsruhe, Karlsruhe, Germany, in 2001.

He is a Principal Researcher with Microsoft Research, Redmond, WA, USA, and a member of Microsoft Quantum—Redmond, Redmond, WA, USA. He was a Senior Research Staff Member with NEC Labs America, Princeton, NJ, USA, from 2005 to 2013, and a Post-Doctoral Fellow with the Institute for Quantum Computing, Waterloo, ON, Canada, from 2003 to 2004. He researched on projects funded by ARO, NSA, the European Union, and the German DFG. From 2011 to 2013, he was the Main PI of the IARPA QCS project TORQUE, a joint effort between Raytheon/BBN Technologies, Cambridge, MA, USA, NEC Labs America, Princeton, NJ, USA, the University of Waterloo, Waterloo, ON, Canada, and the University of Melbourne, Melbourne, VIC, Australia. His current research interests include on quantum algorithms, quantum programming languages, reversible computing, and quantum circuit libraries.

Nathan Wiebe received the Ph.D. degree in physics (quantum computing) from the University of Calgary, Calgary, AB, Canada, in 2011.

He then received a Post-Doctoral fellowship from the Institute for Quantum Computing, University of Waterloo, Waterloo, ON, Canada. He was with the Quantum Architecture and Computing Group, Microsoft Research, Redmond, WA, USA, in 2013. His current research interests include designing quantum algorithms for simulating physical systems and for machine learning as well as developing machine learning methods for characterizing quantum systems. In quantum machine learning, he has provided the first algorithms for deep learning on quantum computers, developed faster quantum methods for training perceptrons, clustering, gradient descent, least squares fitting and boosting as well as inventing the field of quantum Hamiltonian learning.

Giovanni De Micheli (M'83–SM'89–F'94) received the Nuclear Engineer degree from the Politecnico di Milano, Milan, Italy, in 1979 and the M.S. and Ph.D. degrees in electrical engineering and computer science from the University of California at Berkeley, Berkeley, CA, USA, in 1980 and 1983, respectively.

He was a Professor of electrical engineering with Stanford University, Stanford, CA, USA. He is a Professor and the Director with the Institute of Electrical Engineering, EPF Lausanne, Lausanne, Switzerland. His current research interests include several aspects of design technologies for integrated circuits and systems, such as synthesis for emerging technologies, networks on chips, and 3-D integration.

Prof. De Micheli was a recipient of the 2016 IEEE/CS Harry Goode Award for seminal contributions to design and design tools of Networks on Chips, the 2016 EDAA Lifetime Achievement Award, the 2012 IEEE/CAS Mac Van Valkenburg Award for contributions to theory, practice, and experimentation in design methods and tools, and the 2003 IEEE Emanuel Piore Award for contributions to computer-aided synthesis of digital systems, and the D. Pederson Award for the Best Paper on the IEEE Transactions on CAD/ICAS in 1987 and 2018. He is a fellow of ACM, a member of the Academia Europaea, and an International Honorary Member of the American Academy of Arts and Sciences.