Cofactor Sharing for Reversible Logic Synthesis

Alireza Shafaei, University of Southern California
Mehdi Saeedi, University of Southern California
Massoud Pedram, University of Southern California

Improving circuit realization of known quantum algorithms by CAD techniques has benefits for quantum experimentalists. In this paper, the problem of synthesizing a given function on a set of ancillea is addressed. The proposed approach benefits from extensive sharing of cofactors among cubes that appear on function outputs. Accordingly, it can be considered as a multi-level logic optimization technique for reversible circuits. In particular, the suggested approach can efficiently implement any n-input, m-output lookup table (LUT) by a reversible circuit. This problem has interesting applications in the Shor's number-factoring algorithm and in quantum walk on sparse graphs. Simulation results reveal that the proposed cofactor-sharing synthesis algorithm has a significant impact on reducing the size of modular exponentiation circuits for Shor's quantum factoring algorithm, oracle circuits in quantum walk on sparse graphs, and the well-known MCNC benchmarks.

Categories and Subject Descriptors: B.6 [**Hardware**]: Logic Design

General Terms: Hardware, Logic Design, Design Aids, Automatic Synthesis

Additional Key Words and Phrases: Reversible Logic, Logic Synthesis, Cofactor Sharing

ACM Reference Format:
Alireza Shafaei, Mehdi Saeedi, and Massoud Pedram, 2013. Cofactor Sharing for Reversible Logic Synthesis. *ACM J. Emerg. Technol. Comput. Syst.* 0, 0, Article 0 (August 2014), 19 pages.
DOI:http://dx.doi.org/10.1145/0000000.0000000

1. INTRODUCTION

Quantum information processing has captivated atomic and optical physicists as well as theoretical computer scientists by promising a model of computation that can improve the complexity class of several challenging problems [Nielsen and Chuang 2000]. A key example is Shor's quantum number-factoring algorithm which factors a semiprime M with complexity $O((\log M)^3)$ on a quantum computer. The best-known classical factoring algorithm, the *general number field sieve*, needs $O(e^{(\log M)^{1/3}(\log\log M)^{2/3}})$ time complexity. Other quantum algorithms with superpolynomial speedup on a quantum computer include quantum algorithms for discrete-log, Pell's equation, and walk on a binary welded tree [Bacon and van Dam 2010].

This research was supported in part by the Intelligence Advanced Research Projects Activity (IARPA) via Department of Interior National Business Center contract number D11PC20165. The U.S. Government is authorized to reproduce and distribute reprints for Governmental purposes notwithstanding any copyright annotation thereon. The views and conclusions contained herein are those of the authors and should not be interpreted as necessarily representing the official policies or endorsements, either expressed or implied, of IARPA, DoI/NBC, or the U.S. Government.
Author's addresses: A. Shafaei, M. Saeedi, and M. Pedram, Department of Electrical Engineering, University of Southern California, Los Angeles, CA 90089.
A preliminary version of this paper has been appeared in [Shafaei et al. 2013].
Permission to make digital or hard copies of part or all of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies show this notice on the first page or initial screen of a display along with the full citation. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, to republish, to post on servers, to redistribute to lists, or to use any component of this work in other works requires prior specific permission and/or a fee. Permissions may be requested from Publications Dept., ACM, Inc., 2 Penn Plaza, Suite 701, New York, NY 10121-0701 USA, fax +1 (212) 869-0481, or permissions@acm.org.
© 2014 ACM 1550-4832/2014/08-ART0 $15.00
DOI:http://dx.doi.org/10.1145/0000000.0000000

Improving circuit realization of known quantum algorithms — the focus of this work — is of a particular interest for lab experiments. In 2000, Vandersypen et al. [2000] implemented Shor's quantum number-factoring algorithm to factor the number 15. In March 2012, physicists published the first quantum algorithm that can factor a three-digit integer, 143 [Xu et al. 2012]. CAD algorithms and tools are required to help with physical circuit realization even for a few number of qubits and gates. For example, a previous method in [Schaller and Schützhold 2010] required at least 14 qubits to factor 143. This exceeds the limitation of current quantum computation technologies. Hence, Xu et al. [2012] introduced an optimization approach to reduce the number of total qubits.

In this paper, we propose an automatic synthesis algorithm that uses *cofactor-sharing* to synthesize quantum circuits that have applications in, at least, quantum circuits for number factoring and quantum walk [Childs et al. 2003]. In particular, we aim to synthesize a given *lookup table* (LUT) by reversible gates. Following [Markov and Saeedi 2012], an (n, m)-lookup table takes n read-only inputs and $m > \log_2 n$ zero-initialized ancillae (outputs). For each 2^n input combination, an (n, m)-LUT produces a pre-determined m-bit value. Markov and Saeedi [2012] showed LUT synthesis can improve modular exponentiation circuits for Shor's algorithm. In this paper, we generalize the idea of LUT synthesis by extensive sharing of cofactors, and use the shared cofactors to improve the practical implementation of given functions. Among different applications, we particularly focus on quantum walk on graphs and Shor's quantum-number factoring algorithms. In addition, we discuss how cofactor-sharing synthesis can improve cost of irreversible benchmarks. It is worth noting that the presented algorithm can also be considered as a general synthesis approach given that any n-input, m-output Boolean function can be implemented by an (n, m)-LUT, providing enough number of ancillae.

The rest of the paper is organized as follows. In Section 2, basic concepts are introduced. Section 3 highlights a number of applications for LUT synthesis. Related works are discussed in Section 4. We propose our extensive cofactor-sharing algorithm and LUT synthesis approach in Section 5. Experimental results are given in Section 6, and finally Section 7 concludes the paper.

2. BASIC CONCEPT

In this section, we review concepts required to understand the rest of this paper. For more information on reversible logic synthesis, please refer to the recent survey by Saeedi and Markov [2013].

2.1. Boolean Logic

The set of n variables of a Boolean function is denoted as $x_0, x_1, \ldots, x_{n-1}$. For a variable x, x and $\bar{x}$ are *literals*. A Boolean product, *cube*, is a conjunction (AND) of literals where x and $\bar{x}$ do not appear at the same time. A *minterm* is a cube in which each of the n variables appear once, in either its complemented or un-complemented form. A *sum-of-product* (SOP) Boolean expression is a disjunction (OR) of a set of cubes. An *exclusive-or-sum-of-product* (ESOP) representation is an XOR (modulo-2 addition) of a set of cubes. For a given Boolean function $f(x_0, \ldots, x_i, \ldots, x_{n-1})$, the *cofactor* of f with respect to literal x_i is $f(x_0, \ldots, 1, \ldots, x_{n-1})$, and with respect to literal $\bar{x_i}$ is $f(x_0, \ldots, 0, \ldots, x_{n-1})$. For a finite set A, a one-to-one and onto (bijective) function $f : A \to A$ is a *permutation*, which is called a *reversible function*. To convert an irreversible specification to a reversible function, input/output must be added.

2.2. Quantum Computing

Quantum Bit and Register. A quantum bit, *qubit*, can be treated as a mathematical object that represents a quantum state with two basic states $|0\rangle$ and $|1\rangle$. It can also carry a linear combination $|\psi\rangle = \alpha|0\rangle + \beta|1\rangle$ of its basic states, called a *superposition*, where α and β are complex numbers and $|\alpha|^2+|\beta|^2=1$. Although a qubit can carry any norm-preserving linear combination of its basic states, when a qubit is *measured*, its state collapses into either $|0\rangle$ or $|1\rangle$ with probabilities $|\alpha|^2$ and $|\beta|^2$, respectively. A *quantum register* of size n is an ordered collection of n qubits. Apart from the measurements that are commonly delayed until the end of a computation, all quantum computations are reversible.

Quantum Gates and Circuit. A matrix U is *unitary* if $UU^{\dagger} = I$ where $U^{\dagger}$ is the conjugate transpose of U and I is the identity matrix. An n-qubit *quantum gate* is a device which performs a $2^n \times 2^n$ unitary operation U on n qubits in a specific period of time. For a gate g with a unitary matrix U_g, its inverse gate g^{-1} implements the unitary matrix U_g^{-1}. A reversible gate/operation is a 0-1 unitary, and reversible circuits are those composed with reversible gates. A reversible gate realizes a reversible function. A *multiple-control Toffoli* (MCT) gate $\mathrm{C}^n\mathrm{NOT}$ $(x_1, x_2, \cdots, x_{n+1})$ passes the first n qubits unchanged. These qubits are referred to as *controls*. This gate flips the value of $(n+1)$st qubit if and only if the control lines are all one (positive controls). Therefore, the action of the multiple-control Toffoli gate may be defined as follows: $x_{i(out)} = x_i (i < n + 1), x_{n+1(out)} = x_1 x_2 \cdots x_n \oplus x_{n+1}$. Negative controls may be applied similarly. For $n = 0$, $n = 1$, and $n = 2$ the gates are called NOT, CNOT, and Toffoli, respectively. The lines which are added to make an irreversible specification reversible are named *ancillae*, which normally start with the state $|0\rangle$. The zero-initialized ancillae may be modified inside a given subcircuit, but should be returned to zero at the end of computation to be reused.

Cost Model. Quantum cost (QC) is the number of NOT, CNOT, and controlled square-root-of-NOT gates required for implementing a given reversible function. QC of a circuit is calculated by a summation over the QCs of its gates. In addition to the QC model, a single-number cost based on the number of two-qubit operations required to simulate a given gate was proposed by Maslov and Saeedi [2011]. This model captures the complexity of physical implementation of a given gate based on the Hamiltonian describing the underlying quantum physical system. In particular, it estimates the cost of a $\mathrm{C}^n\mathrm{NOT}$ (and $n \geq 2$) as $2n - 5$ 3-qubit Toffoli gates (and $10n - 15$ 2-qubit gates).

3. APPLICATIONS IN QUANTUM COMPUTING

Specific reversible circuits must be motivated by applications. In the following, we introduce several immediate applications of cofactor-sharing synthesis in quantum computation. In these applications, ancillea are used as outputs. In other non-immediate applications, one should add ancillea to use the cofactor-sharing.

3.1. Quantum Algorithm for Number Factoring

Shor's quantum number factoring uses the quantum circuit for modular exponentiation $b^x\%M$ (% is modulo operation) for a randomly selected number b, and a semiprime $M = pq$, for primes p and q. Modular exponentiation is performed by n conditional modular multiplications $Cx\%M$ where C and M are coprime. Precisely, for the binary expansion $x = x_n 2^n + x_{n-1}2^{n-1} + \cdots + x_0$ (and x_i is 0 or 1), $b^x\%M = b^{x_n 2^n} \times b^{x_{n-1}2^{n-1}} \times \cdots \times b^{x_0}\%M$. Hence, one needs to implement multiplication by $b^{2^n}\%M$ conditioned on x_n, multiplication by $b^{2^{n-1}}\%M$ conditioned on $x_{n-1}, \ldots,$ and multiplication by $b\%M$ conditioned on x_0, in sequence.

Markov and Saeedi [2012, Section 7.2] introduced an (n,m)-LUT (for $n=4$) to implement the (four) most expensive conditional modular multiplications that appear in modular exponentiation to reduce the total cost. As an example [Markov and Saeedi 2012, Figure 15] implements conditional modular multiplications by 4, 16, 82, and 25 in modular exponentiation for $b=2$, $M=87=3\times 29$ by a systematic method. The related outputs of this (4,7)-LUT are 1, 4, 16, 64, 82, 67, 7, 28, 25, 13, 52, 34, 49, 22, 1, and 4 which are obtained by considering different combinations (by multiplication) of 4, 16, 82, and 25 %87. Except for the four most expensive modular multiplications, other modular multiplications are implemented directly in [Markov and Saeedi 2012]. In this work, however, we propose an automatic synthesis method that can further improve modular exponentiation circuits.

3.2. Quantum Walk for Sparse Graphs

In [Chiang et al. 2010, Thereom 1], the authors proposed a polynomial-size circuit for quantum walk on a sparse graph with 2^n nodes along with an adjacency matrix P. A graph is *sparse* if each node has at most d transitions (or edges) to other nodes. To propose the circuit, the authors assumed **(1)** there is a polynomial-size reversible circuit returning the list of (at most d) n-bit neighbors of the node x according to P **(2)** there is a polynomial-size reversible circuit returning the list of (at most d) t-bit precision transition probabilities. Our cofactor-sharing synthesis can be used to construct circuits for (1) and (2).

3.3. Quantum Walk on Binary Welded Tree

As a special case of quantum walk on sparse graphs, one can consider a binary welded tree. A *binary welded tree* (BWT) is a graph which consists of two binary trees that are *welded* together with a random function between the leaves. Figure 1(a) shows a sample BWT. In a BWT every node has degree three except the root of each tree (which has degree two). A BWT has $2(2^{n+1}-1)$ nodes for a binary tree of height n. Therefore, strings of $m > \lceil \log_2 2(2^{n+1}-1)\rceil$ bits are required to represent each node uniquely (minimum m is $n+2$). All edges of a node in a BWT are uniquely colored and each color is denoted by c. The number of colors used in a BWT is at least 3 and at most 4 (by Vizing's theorem for graph coloring).

In [Childs et al. 2003], an *oracle-based* quantum walk algorithm on BWT has been proposed which is exponentially faster, with $O(n)$ oracle queries, on a quantum computer than on a classical computer. The best-known classical algorithm needs O(2^n) oracle queries. For any edge color c, the oracle function $v_c(a)$ takes as input the node label a, and returns the label of a node that is connected to node a with a c color edge. As an example for the BWT in Figure 1(a) and c=black, we have (Figure 1(b)) $v_c(7)=16, v_c(8)=17, v_c(9)=15, v_c(11)=19, v_c(12)=22, v_c(13)=18, v_c(14)=20$ (and vice versa, e.g., $v_c(16)=7$).[1] If there is no connection to a with color c, the oracle returns the unique label `invalid`. In [Childs et al. 2003], this unique value is all ones. Outputs should be constructed on a separate register so that input register remains unchanged for future queries. Note that in a physical implementation, besides the number of queries to the oracle, the computation performed by the oracle also affects the runtime. Accordingly, we use cofactor-sharing synthesis to improve the physical implementation of a given oracle circuit.

[1]Permutations in BWT include 2-cycles. For a synthesis algorithm that extensively works with cycles see [Saeedi et al. 2010].

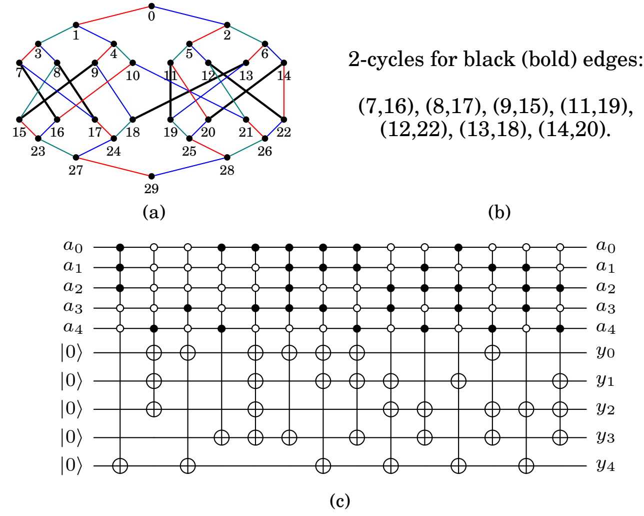

Fig. 1. (a) A sample binary welded tree. (b) Lookup table of the oracle for black edges. A 2-cycle (a, b) is a permutation which exchanges two elements and keeps all others fixed. (c) An oracle implementation. In general, one needs l C^{k}NOT gates to implement each minterm where l is the number of bits with value 1 in the binary representation of the minterm. For example, the first gate implements 16 (i.e., "10000" in binary) for 7 (i.e., "00111" in control lines — two negative and three positive controls). The second gate implements 7 (i.e., "00111" which needs three target lines) for 16 (i.e., "10000" in control lines). Other gates can be constructed similarly.

4. RELATED WORK

A trivial approach to synthesize an LUT is to implement each input combination of an (n, m)-LUT with at most m C^{n}NOT gates. For example, reconsider the BWT in Figure 1(a) where the circuit in Figure 1(c) constructs the oracle. To handle the INVALID label, initialize outputs to all ones and flip target locations in Figure 1(c). However, large number of Toffoli gates with many controls are expensive for physical implementation.

ESOP-based approaches [Fazel et al. 2007; Nayeem and Rice 2011] are fast and are able to handle large sizes of both reversible and irreversible functions. The basic idea is to write each output as an ESOP representation and implement each term by a multiple-control Toffoli gate [Fazel et al. 2007]. In recent years, several improved ESOP-based approaches, e.g., [Nayeem and Rice 2011], have been proposed which use shared product terms (cubes) to reduce the number of Toffoli gates. However, these approaches usually lead to expensive multiple-control Toffoli gates with many controls.

Reversible logic synthesis methods [Saeedi and Markov 2013] can also be used to synthesize a given (n, m)-LUT. To this end, input register should be copied (by m CNOT gates) into output register so that inputs remain unchanged. However, these approaches are general and may not exploit LUT structures for cost reduction. Other approaches are based on Davio decompositions[2] which include the method in [Markov and Saeedi 2012] for $(4, m)$-LUT synthesis and the method in [Wille and Drechsler 2009]. Method in [Markov and Saeedi 2012] uses cofactors for multi-level optimization in logic synthesis but it is limited to $(4, m)$-LUT implementation. By assuming that the factors have already been computed on dedicated ancillae, [Wille and Drechsler 2009] implements the Davio decompositions. It leads to numerous ancillae.

[2]Positive Davio and negative Davio decompositions are defined by $f = f_{x_i=0} \oplus x_i.f_{x_i=2}$ and $f = f_{x_i=1} \oplus \bar{x}_i.f_{x_i=2}$ for $f_{x_i=2} = f_{x_i=0} \oplus f_{x_i=1}$.

5. PROPOSED SYNTHESIS ALGORITHM

Multi-level logic synthesis for irreversible functions has a rich history [Brayton et al. 1984]. However, conventional logic-synthesis approaches cannot be immediately used for cofactor extraction and multi-level circuit realization in reversible circuits. Basically, in a multi-level implementation of a set of functions, it is allowed to use an *unlimited* number of intermediate signals. This is due to the fact that intermediate signals in classical circuits can be realized with low cost. However, in quantum circuits intermediate signals should be constructed on qubits[3] and the number of qubits in current quantum technologies is very *limited*. Therefore, appropriate modification to the existing approaches is essential.

In this section, we propose a synthesis algorithm which is equipped with techniques to reduce the number of ancillae required in a multi-level logic optimization. The section begins with the description of the input function. Sharing methods based on cubes as well as cofactors are presented next. A lookahead search is then discussed which explores various ordering of cofactors to find one that minimizes the synthesis cost. This method is based on a variant of unate covering problem. Finally, we exploit the trade-off between cost and ancillae by adding more ancilla lines to decrease the implementation cost of large cubes (i.e., cubes with large number of control lines).

5.1. Input Specification

A reversible synthesis problem intends to implement a given n-input, m-output Boolean function by a reversible circuit. We assume that the input function is given in the ESOP format. That is, $y_j = c_{0,j} \oplus c_{1,j} \oplus \cdots \oplus c_{k_j-1,j}$, for $0 \leq j < m$, where y_j and $c_{k,j}$ denote an output variable and a cube, respectively, and k_j is the number of cubes in y_j. The input function is stored as a list of cubes, called `cube_list`, in which a cube $\mathcal{C}$ in an n-input, m-output Boolean function with input variables x_i and output variables y_j ($0 \leq i < n$, $0 \leq j < m$) is represented as a row vector $[\alpha_0, \ldots, \alpha_{n-1}, \beta_0, \ldots, \beta_{m-1}]$ [Brayton et al. 1984, Section 2.3]. In this notation, $\alpha_i = 0$ if x_i appears negatively, $\alpha_i = 1$ if x_i appears positively, and $\alpha_i = 2$ if x_i does not appear in $\mathcal{C}$. Hence, number of $\alpha_i \neq 2$ ($0 \leq i < n$) denotes the number of literals in $\mathcal{C}$. Additionally, $\beta_j = 0$ if $\mathcal{C}$ is not available in y_j, and $\beta_j = 1$ if $\mathcal{C}$ is available in y_j. Accordingly, number of $\beta_j \neq 1$ ($0 \leq j < m$) specifies number of outputs that need $\mathcal{C}$.

Example 5.1. Consider the `f2_158` benchmark which is a 4-input, 4-output Boolean function with the following ESOP representation (generated by the EXORCISM-4 [Mishchenko and Perkowski 2001]). Its `cube_list` is shown in Table I. This benchmark is used in the next sections to demonstrate the proposed approach.

$$
\begin{aligned}
y_0 &= x_0x_1'x_2x_3' \oplus x_0'x_2 \oplus x_0'x_1x_2x_3 \\
y_1 &= x_0x_1'x_2' \oplus x_0x_1'x_2'x_3 \oplus x_0x_2'x_3' \oplus x_0'x_1x_2x_3 \oplus x_0'x_1 \\
y_2 &= x_0x_1'x_2'x_3 \oplus x_0x_2'x_3' \oplus x_0x_1'x_2x_3' \\
y_3 &= x_0x_1'x_2'x_3 \oplus x_0'x_3 \oplus x_0'x_1x_2x_3
\end{aligned}
$$

5.2. Cube Sharing

A cube $\mathcal{C}$ that contains $p \leq n$ literals and is required by $q \leq m$ outputs can be constructed by q MCT gates each of which has p controls and a target on one of the outputs. The polarity of each control line is matched with the polarity of its corresponding literal in $\mathcal{C}$. Accordingly, for don't care literals no control line is added. As an example,

[3]Recall that reversible functions are unitary transformation. As a result, explicit fanouts and loops/feedback are prohibited.

Table I. `cube_list` for `f2_158` benchmark. See Example 5.1.

Cube $\mathcal{C}$	α_0	α_1	α_2	α_3	β_0	β_1	β_2	β_3
$c_0 = x_0x_1'x_2'$	1	0	0	2	0	1	0	0
$c_1 = x_0x_1'x_2'x_3$	1	0	0	1	0	1	1	1
$c_2 = x_0x_2'x_3'$	1	2	0	0	0	1	1	0
$c_3 = x_0x_1'x_2x_3'$	1	0	1	0	1	0	1	0
$c_4 = x_0'x_2$	0	2	1	2	1	0	0	0
$c_5 = x_0'x_3$	0	2	2	1	0	0	0	1
$c_6 = x_0'x_1x_2x_3$	0	1	1	1	1	1	0	1
$c_7 = x_0'x_1$	0	1	2	2	0	1	0	0

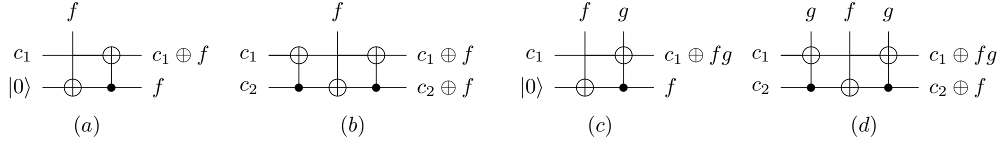

Fig. 2. Copying a cube by at most two CNOT gates (a) with and (b) without a zero-initialized ancilla. Similarly, copying a cofactor by at most two CNOT gates (c) with and (d) without a zero-initialized ancilla.

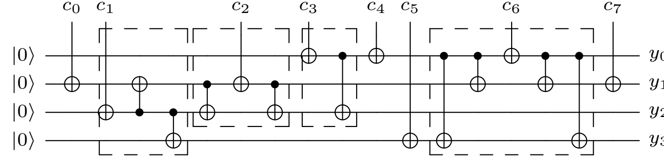

Fig. 3. Circuit for `f2_158` after applying the cube sharing method. Constructing each cube individually results in cost of $3 \times 5 + 3 \times 13 + 8 \times 26 = 262$. On the other hand, cube sharing reduces the cost to $9 \times 1 + 3 \times 5 + 2 \times 13 + 3 \times 26 = 128$.

cube $c_2 = [1, 2, 0, 0, 0, 1, 1, 0]$ can be realized by two MCT gates C^{3}NOT(x_0, x_2', x_3', y_1), C^{3}NOT(x_0, x_2', x_3', y_2).

To avoid multiple constructions of the same cube, as done in [Nayeem and Rice 2011], common cubes among different functions may be shared. This can be performed by constructing the shared cube once and copying the result by several CNOTs. However, this cube sharing method requires an appropriate mechanism for copying cubes on output lines which is described next.

Contents of a cube that is constructed on an output line with any arbitrary Boolean value can be copied to other outputs by at most two CNOT gates. Consider the circuit of Figure 2(a) which has two outputs with initial values c_1 and $|0\rangle$. Assume that f is a cube (its actual circuit is not shown) and the goal is to construct $c_1 \oplus f$ on the first qubit. Here, only one CNOT is needed. Now, assume that the value in the second qubit is any arbitrary Boolean value c_2. To remove the effect of c_2, we add one extra gate before constructing the cofactor f, as shown in Figure 2(b).

As an example, applying the cube sharing method on `f2_158` benchmark leads to the circuit shown in Figure 3. Dashed boxes highlight cases where a cube is needed by more than one output. Different ordering of cubes may change the number of copying CNOTs.

5.3. Cofactor Sharing

Cube sharing can reduce the number of MCT gates, but it leaves the number of controls as is. Recent ESOP-based methods for reversible circuits, e.g., [Nayeem and Rice 2011], restrict circuit optimization to use only cubes of the ESOP representation of the input function, which can limit their performance. For example, consider $y_0 = ab$ and $y_1 = abc$. Note that each cube appears once. Therefore, no cube can be shared. Figure 4(a) shows a circuit with one C^{2}NOT and one C^{3}NOT. However, relaxing the constraint

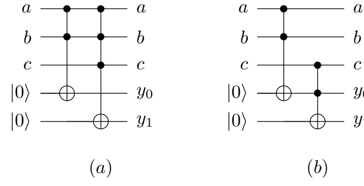

Fig. 4. Circuits for $y_0 = ab, y_1 = abc$, (a) without cofactor sharing, (b) with cofactor sharing.

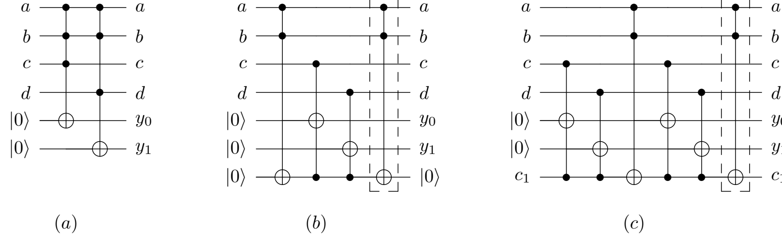

Fig. 5. Circuits for function $y_0 = abc, y_1 = abd$. (a) Initial circuit. (b) An equivalent circuit constructed by reusing ab as a shared cofactor. (c) When no unused zero-initialized output exists. Gates in dashed box are used to un-compute the cofactor ab.

of sharing available cubes promises a significant cost reduction. As an example, it is possible to reuse the cofactor ab twice. This can be done by constructing the cofactor ab on y_0, and reusing it to construct abc on y_1 (Figure4(b)).

Cofactor ab in the function of Figure 4 was also needed as a cube by an output line. However, in cases where this is not valid (i.e. cofactor is not required by any of the outputs), a zero-initialized ancilla can be employed to temporarily construct the cofactor, and then use it to optimize different cubes. This process should be followed by *un-computing* the constructed cofactor to recover the zero-initialized ancilla for future use. The reason for un-computation is twofold. **(1)** Without un-computation, each cofactor needs a new ancilla (qubit) and the number of available qubits is very restricted in current quantum technologies. **(2)** Constructing a zero state from an unknown quantum state *generally* needs an exponential number of gates [Plesch and Brukner 2011].

Hence, a shared cofactor can be constructed on a zero-initialized ancilla by an MCT gate M. To reuse the ancilla, one needs to un-compute the constructed cofactor by applying M at the end of computation. As an example, consider $y_0 = abc$, $y_1 = abd$. Figure 5(a) shows the circuit. However, as shown in Figure 5(b), one can temporarily construct the cofactor ab on a zero-initialized ancilla (the first gate). Afterwards, based on Figure 2-(c), we can use the cofactor to implement dependent cubes (gates #2 & #3). The constructed cofactor is un-computed finally. Even an output line with any arbitrary Boolean value can be used to construct the cofactor by following the circuit of Figure 2(d). An example is shown in Figure 5(c).

To enable cofactor sharing, a list of cofactors that are shared between at least two cubes is initially created. For k cubes, the maximal shared cofactors between all cubes can be found by at most k^2 comparisons. Shared cofactors are stored by another tabular format, called `shared_cofactor_list`, which additionally keeps for each shared cofactor all of its dependent cubes (i.e. cubes that contain the cofactor), the amount of cost reduction gained by sharing this cofactor, and a Boolean variable which determines whether the cofactor also appears as a cube.

Table II reports the `shared_cofactor_list` of `f2_158` benchmark. Cost reduction values are described in the next section.

Table II. shared_cofactor_list for f2_158 benchmark (Example 5.1)

Shared cofactor	Dependent cubes	Cost reduction value	Cube?
$s_0 = x_0x'_3$	$\{c_2, c_3\}$	11	No
$s_1 = x'_0x_3$	$\{c_5, c_6\}$	13	Yes
$s_2 = x_0x'_1x'_2$	$\{c_0, c_1\}$	21	Yes
$s_3 = x_0x'_2$	$\{c_0, c_1, c_2\}$	19	No
$s_4 = x'_0x_2$	$\{c_4, c_6\}$	13	Yes
$s_5 = x_0x'_1$	$\{c_0, c_1, c_3\}$	24	No
$s_6 = x'_0x_1$	$\{c_6, c_7\}$	13	Yes

5.4. Implementing a Shared Cofactor

The cost of implementing a given shared cofactor along with its dependent cubes is computed by Algorithm 1 in our proposed synthesis method. The circuit that realizes the shared cofactor and its dependent cubes is obtained by the same algorithm as well. Among inputs to the algorithm, $output_status$ is a bitmap whose ith index is set if the value of output line i is still zero. Also, $scof_cost$ denotes the quantum cost of the MCT gate that will realize the shared cofactor, $scof_cnots$ is the number of CNOTs needed for copying the shared cofactor on corresponding output lines when the shared cofactor is a cube itself, and $scof_controlNum$ indicates the number of control lines in the shared cofactor. Similarly, $cubes_cost$ is the sum of quantum costs of dependent cubes, $cubes_cnots$ denotes the total number of CNOTs needed for copying each dependent cube, and $cube_controlNum$ is the number of control lines in a dependent cube.

Additionally, `CountLiterals()` is a function that returns the number of literals in a given product term, which is equivalent to the number of control lines in the respective MCT gate. This number is the input to the `FindMCTCost()` function which in turn computes the cost of the MCT gate. Based on the current status of output lines, `FindCopyingCost()` calculates the number of CNOT gates needed for copying contents of a cube on required output lines, and updates the $output_status$ accordingly. Furthermore, `checkEmptyOutputs()` sets Boolean variable $emptyOutput$ to true if a zero-initialized output line that is not going to be used in this step (i.e., a cube will not be constructed on it) is available. Boolean variable $emptyOutputForSj$ will also be true if `checkEmptyOutputsOfSj()` can find a zero-initialized output line used by s_j which will not be used by other dependent cubes of s_j.

Algorithm 1 initially finds the quantum cost of the shared cofactor. The number of copying CNOTs are also calculated when the shared cofactor is itself a cube. Then, dependent cubes are constructed based on the cube sharing method. However, each dependent cube uses the intermediate value of the shared cofactor, and thus an MCT gate with $cube_controlNum - scof_controlNum + 1$ control lines is needed. Copying CNOT gates for each dependent cube are also added.

If the shared cofactor is not a cube, the cost of un-computing the shared cofactor is added as well. Moreover, the shared cofactor is constructed on a zero-initialized output line which will not be used by any other dependent cubes. However, if no such output can be found, an ancilla line is needed (c.f. condition [1] in Algorithm 1).

On the other hand, a shared cofactor that also appears as a cube is implemented on a zero-initialized output which is needed by itself, but not by any other dependent cubes. In case that such output line is not available, shared cofactor is constructed on the ancilla line (c.f. condition [2] in Algorithm 1). Contents of ancilla are then copied by CNOT gates to output lines that need the shared cofactor. Accordingly, copying CNOTs that we added earlier are excluded from the cost.

In Algorithm 1, the order in which dependent cubes are implemented can affect final cost. For simplicity, Algorithm 1 randomly picks dependent cubes. However, synthesis cost can be improved by exploring various orderings, and selecting the one that results

ALGORITHM 1: Shared Cofactor Cost Computation

Input: A shared cofactor, s_j, the `cube_list`, and the initial status of output lines, $output_status$.
Output: The cost of implementing s_j together with all of its dependent cubes.
$scof_cnots$ = 0; $cubes_cost$ = 0; $cubes_cnots$ = 0;
$scof_controlNum$ = CountLiterals(s_j);
$scof_cost$ = FindMCTCost($scof_controlNum$);
if *(s_j is a cube)* **then**
    $emptyOutputForSj$ = checkEmptyOutputsOfSj(s_j, $output_status$);
    $scof_cnots$ = FindCopyingCost(s_j, $output_status$);
else
    $emptyOutput$ = checkEmptyOutputs($output_status$);
end
for *each s_j's dependent cube c_i* **do**
    if *($c_i \neq s_j$)* **then**
        $cube_controlNum$ = CountLiterals(c_i);
        $cubes_cost$ += FindMCTCost($cube_controlNum - scof_controlNum + 1$);
        $cubes_cnots$ += FindCopyingCost(c_i, $output_status$);
    end
end
$cost$ = $scof_cost$ + $scof_cnots$ + $cubes_cost$ + $cubes_cnots$;
if *(s_j is not a cube)* **then**
    $cost$ += $scof_cost$; `// adding uncomputatopn cost of` s_j
`[1]:`   **if** *(!$emptyOutput$)* **then** Construct s_j on the ancilla line;
else
`[2]:`   **if** *(!$emptyOutputForSj$)* **then**
        Let n_j be the number of output lines that need s_j as a cube;
        $cost$ += $scof_cost + n_j - scof_cnots$;
        Construct s_j on the ancilla line;
    end
end
return $cost$

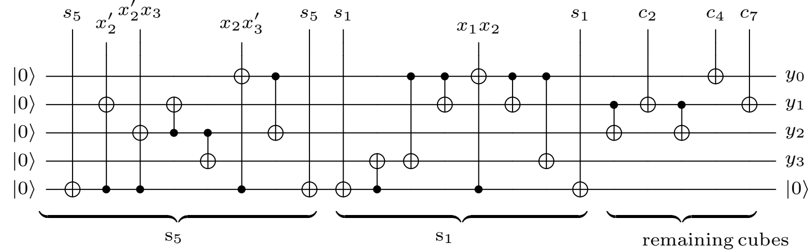

Fig. 6. Circuit of `f2_158` after applying the cofactor-sharing method using $s_5 \rightarrow s_1$ as the sequence of selected shared cofactors. The cost is $10 * 1 + 7 * 5 + 4 * 13 = 97$.

in the minimum cost. This can be achieved by applying an exhaustive or a lookahead search (Section 5.5) with the penalty of runtime.

Example 5.2. Applying Algorithm 1 on s_5 followed by s_1 in `f2_158` circuit is shown in Figure 6. For s_5, condition [1] becomes valid as all zero-initialized output lines will be used in this step. The ancilla line is thus used. On the other hand, since all output lines have a non-zero value, condition [2] will be true for s_1. The ancilla line was uncomputed in the preceding step, and hence can be used once again.

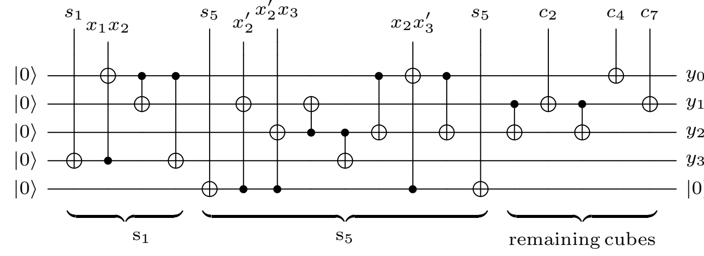

Fig. 7. Circuit of f2_158 after applying the cofactor-sharing method using $s_1 \rightarrow s_5$ as the sequence of selected shared cofactors. The cost is $8*1+6*5+4*13=90$.

5.5. Lookahead Search

Assume that we have k cubes, $c_1, \ldots, c_k$, which produce l shared cofactors, $s_1, \ldots, s_l$. The synthesis problem then intends to find an ordering of shared cofactors such that the synthesis cost of the circuit is minimized. For this purpose, we can create a covering matrix where shared cofactors denote columns, cubes represent rows, and a '1' is inserted in column j, row i if shared cofactor s_j covers (is contained in) cube c_i. The problem looks similar to the *unate covering problem* (UCP) which is used in two-level logic minimization to find a subset of columns (prime implicants) such that all rows (minterms) are covered by at least one column and the cost is minimized [Coudert 1994]. However, it differs from the UCP in the sense that the order in which columns (shared cofactors) are selected affects the cost. In addition, all rows (cubes) may not be covered by the selected ordering of shared cofactors, and remaining uncovered cubes are realized by the cube sharing method.

Example 5.3. Figure 7 depicts the circuit obtained by applying Algorithm 1 on s_1 followed by s_5. As can be seen, by changing the order of shared cofactors, synthesis cost is reduced compared to the circuit of Figure 6.

UCP is optimally solved by a branch-and-bound algorithm, where a column C is initially chosen as the root node according to a cost function. Two subtrees are then generated, one by including C in the final solution and the other by eliminating it from the solution set, which are in turn solved recursively. The final minimal solution is the minimum of the two subtrees. Unfortunately, this approach cannot explore the order of columns and hence may not lead to the minimal solution in cofactor sharing problem. As a result, a lookahead search is rather used for this problem.

Prior to begin the lookahead search, columns are sorted based on a cost function such that those columns that have a higher chance to be included in the final solution are definitely visited. Index of columns are consequently updated based on the sort result. The lookahead search with depth d and maximum node degree Δ then initiates by creating a root node labeled r. A sample lookahead search tree with $d=4$ and $\Delta=2$ is illustrated in Figure 8. Afterwards, node r as well as all of its descendant nodes until depth $d-1$ will generate at most Δ different nodes. Child nodes are chosen such that every path from r to any node contains distinct node labels. Finally, the cost of all paths starting from r are calculated (cost of node r is zero), and the first node (without considering node r) of the minimum cost path is selected to be inserted into the final solution order. The same process is re-executed for the next selections, until no more column is left.

For the cofactor sharing problem, after a shared cofactor is selected, its dependent cubes are removed from the cube_list (since they have been covered). Then, other available shared cofactors are checked to see if they still have more than one dependent cube. Cofactors that cannot satisfy this condition are no longer a valid shared cofactor, and hence are removed from the shared_cofactor_list. Updating

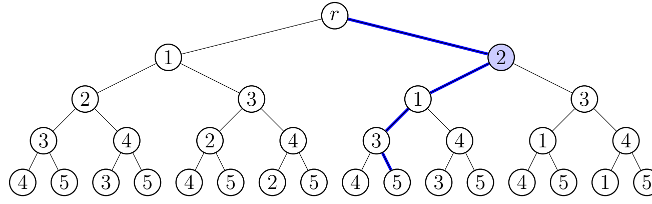

Fig. 8. A lookahead search tree with $d = 4$ and $\Delta = 2$. Assume that the highlighted path $r \to 2 \to 1 \to 3 \to 5$ has the minimum cost among other paths starting from r. Hence, the lookahead search will pick column 2 (the first node after root) for this step.

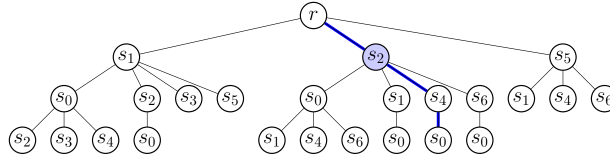

Fig. 9. Lookahead tree for `f2_158` circuit. The minimum cost path is highlighted.

`shared_cofactor_list` is also valuable in terms of pruning branches of the lookahead tree and cutting the runtime.

Example 5.4. In `f2_158` circuit, selecting s_5 which covers cubes c_0, c_1, and c_3 leaves s_0 and s_3 with one, and s_2 with no uncovered dependent cube. Thus, s_0, s_3, and s_2 are no longer a valid shared cofactor, and are not explored in the subtree of s_5. The lookahead tree of `f2_158` circuit is also illustrated in Figure 9. Here, to save space, the root node only explores three shared cofactors s_1, s_2, and s_5.

Steps of the lookahead search used in our proposed synthesis method is presented in Algorithm 2. The algorithm exhaustively traverses a lookahead tree with depth d and maximum node degree Δ in depth-first order. Since a depth-first order is used to search the tree, only one path is traversed at each time and the level of the node that we are currently processing will be denoted by $level$. Consequently, array variables are used to store intermediate results (obtained at different levels) of the current path, which are introduced next. $cost_arr$ and $output_status_arr$ are arrays of size $d+1$ which represent the total cost from root node and the status of zero output lines, respectively. Number of nodes visited at each level as well as the shared cofactor that is selected in the current level are saved in arrays of size d, $visited_nodes_arr$ and $scof_arr$, respectively. Additionally, the cost function which is used to sort `shared_cofactor_list` is set to the amount of cost reduction obtained by sharing each cofactor compared to when only cube sharing is used.

Furthermore, our proposed cofactor-sharing synthesis method is given in Algorithm 3. As mentioned earlier, $output_status$ is responsible to keep track of the status of zero output lines (line 1). Lines 2-3 construct the required lists. Since cofactor sharing is always beneficial in terms of cost reduction, only when no valid shared cofactor is available, the algorithm terminates (line 5). Moreover, line 6 calls Algorithm 2 to select the next cofactor. Lines 7-8 calculate the implementation cost of the selected shared cofactor along with its dependent cubes by calling Algorithm 1, and update the total synthesis cost accordingly. Line 9 removes dependent cubes and invalid shared cofactors from $cube_list$ and $shared_cofactor_list$, respectively. Remaining cubes that were not covered by a shared cofactor are constructed at the end using the method of cube

ALGORITHM 2: Lookahead Search

Input: $shared_cofactor_list$, $cube_list$, $output_status$, lookahead depth, d, and maximum node degree, Δ.
Output: First shared cofactor on the minimum cost path.

```
level = 1; min_cost = ∞; cost_arr[0] = 0; output_status_arr[0] = output_status;
Sort shared_cofactor_list based on cost reduction values in decreasing order;
while ( level > 0 ) do
    cofactor scof = SelectNextSharedCofactor(shared_cofactor_list);
    if ( no shared cofactor is available OR visited_nodes_arr[level] > Δ ) then
        // Backtracking
        visited_nodes_arr[level] = 0;
        level- -;
    else
        visited_nodes_arr[level]++;
        scof_arr[level] = scof;
        output_status_arr[level] = output_status_arr[level − 1];
        cost_arr[level] = cost_arr[level − 1];
        cost_arr[level] += ImplementSharedCofactor(scof, cube_list, outputs_status[level]);
        if ( level < d ) then
            Update cube_list and shared_cofactor_list based on scof;
            level++;
        else
            r_cost = ConstructUncoveredCubes(cube_list, outputs_status_arr[level]);
            if ( cost_arr[level] + r_cost < min_cost ) then
                min_cost = cost_arr[level] + r_cost;
                min_scof = scof_arr[1];
            end
        end
    end
end
return min_scof
```

ALGORITHM 3: Cofactor-Sharing Synthesis

Input: An ESOP-based n-input, m-output Boolean function, f, as well as the lookahead depth, d, and maximum node degree, Δ.
Output: A quantum circuit, which generates f using MCT gates, and its corresponding cost.

```
1:  Define outputs_status as a bitmap of size equal to m;
2:  cube_list = ConstructCubeList(f);
3:  shared_cofactor_list = ConstructSharedCofactorList(cube_list);
4:  total_cost = 0;
5:  while ( shared_cofactor_list is not empty ) do
6:      cofactor scof = LookaheadSearch(shared_cofactor_list, cube_list, outputs_status, d, Δ);
7:      s_cost = ImplementSharedCofactor(scof, cube_list, outputs_status);
8:      total_cost = total_cost + s_cost;
9:      Update cube_list and shared_cofactor_list based on scof;
10: end
11: r_cost = ConstructUncoveredCubes(cube_list, outputs_status);
12: total_cost = total_cost + r_cost;
13: return total_cost
```

sharing (line 11). The result of cofactor-sharing synthesis algorithm on `f2_158` circuit is shown in Figure 10.

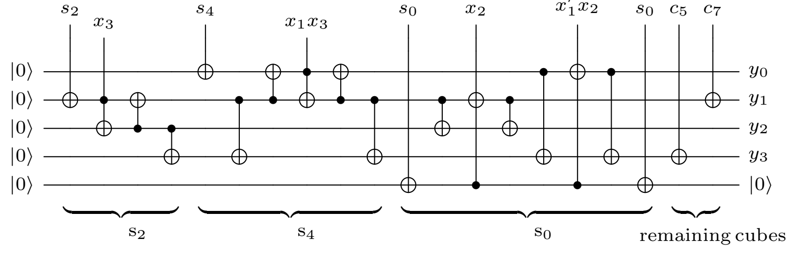

Fig. 10. Circuit of f2_158 after applying the cofactor-sharing method using $s_2 \rightarrow s_4 \rightarrow s_0$ as the sequence of selected shared cofactors. The cost is $10 * 1 + 7 * 5 + 3 * 13 = 84$.

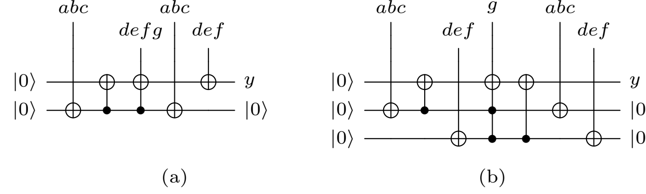

Fig. 11. Implementation of $y = abc \oplus def \oplus abcdefg$ using cofactor-sharing. (a) Cubes could be constructed by only one shared cofactor. Cost is $1 * 1 + 3 * 13 + 1 * 38 = 78$. (b) An extra ancilla line is added so as cube $abcdefg$ can be constructed by two shared cofactors. Cost becomes $2 * 1 + 5 * 13 = 67$.

5.6. Cost and Ancilla Trade-off

When a dependent cube $\mathcal{C}$ with p_c literals is going to be constructed with a shared cofactor with p_s literals, the number of control lines of the MCT gate that realizes $\mathcal{C}$ will be reduced to $p_c - p_s + 1$. This reduction in the number of control lines makes the cofactor-sharing synthesis beneficial in terms of cost optimization. However, $p_c - p_s + 1$ may still be a large number (e.g. if $p_s << p_c$), and thus the cost of the resulting MCT gate would still be high. The problem occurs as we allowed a dependent cube to be built by just one shared cofactor. This is useful in terms of number of ancillae, since only one ancilla is required to temporarily store shared cofactors. However, it may be possible to implement a cube with more than one shared cofactor, which subsequently requires more ancilla (one ancilla per cofactor), but can further reduce the number of control lines of the MCT gate.

Example 5.5. Consider the following 7-input, 1-output function $y = abc \oplus def \oplus abcdefg$. shared cofactors are abc and def. In Figure 11(a) at most one ancilla is allowed. On the other hand, two ancilla lines are used in Figure 11(b) so as cube $abcdefg$ can be constructed by two shared cofactors, improving the cost by 14%.

Shared cofactors that have one or more literals in common can also be used to construct cubes. For instance, if $ab\underline{c}$ was already constructed on y_0, and $\underline{c}d$ on y_1, then $\mathrm{C}^2\mathrm{NOT}(y_0, y_1, y_2)$ realizes $abcd$ on y_2 by using these intermediate variables.

In order to benefit from the cube construction using more than one shared cofactor, we first execute Algorithm 3 to obtain an ordering of shared cofactors $\mathcal{S} = (s_1, \ldots, s_l)$. The following control parameters are also added to the synthesis tool: (1) θ_c which is a threshold value for detecting large cubes, and (2) $AncillaBudget$ which determines the maximum number of available ancilla lines. We then implement shared cofactors $(s_1, \ldots, s_l)$ once again, but this by applying proper modifications to the synthesis algorithm which is explained next.

Assume shared cofactor $s_j \in \mathcal{S}$, $1 \leq j < l$, is going to be implemented. If for any of its dependent cubes c_i, $p_c - p_s + 1 > \theta_c$ then c_i is considered as a large cube. The possibility of constructing c_i using more than one cofactor is investigated next by checking future shared cofactors $(s_{j+1}, \ldots, s_l)$ to see if they are also a shared cofactor for c_i. If these

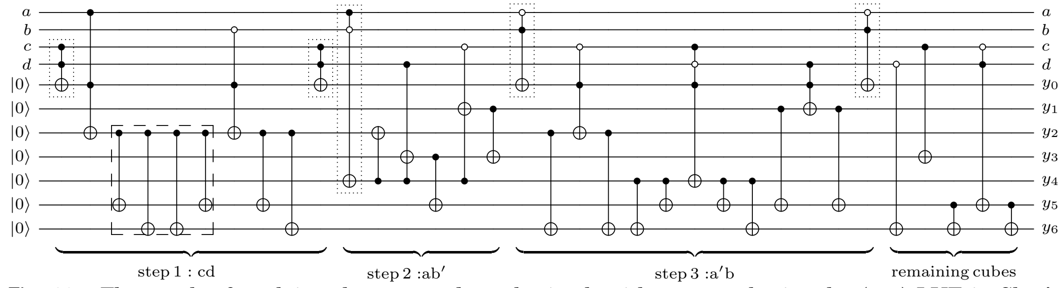

Fig. 12. The result of applying the proposed synthesis algorithm to synthesize the $(4, 7)$-LUT in Shor's algorithm for $M = 65$. The ESOP expansion for outputs can be represented as $y_0 = 0$, $y_1 = ab'c' \oplus a'bd$, $y_2 = ab' \oplus a'bc' \oplus acd \oplus b'cd$, $y_3 = ab'd \oplus ab'c' \oplus c$, $y_4 = ab' \oplus a'bcd'$, $y_5 = ab'd \oplus a'bcd' \oplus acd \oplus b'cd \oplus a'bd \oplus c'd$, $y_6 = d' \oplus a'bc' \oplus a'bcd' \oplus acd \oplus b'cd \oplus c'd$. Shared cofactors are highlighted with dotted boxes. As shown in the dashed box, a post-synthesis optimization can further improve the circuit.

conditions are true, and also a free ancilla line is available (i.e. number of ancillae that currently have a shared cofactor $< AncillaBudget$), the construction of c_i is postponed. Literals of c_i that are common to s_j are also set to 2 (don't care). Furthermore, s_j is stored on an ancilla line. However, when the number of literals in a cube becomes less than θ_c, or no more future shared cofactors can cover the cube, or no free ancilla is left, the postponed cube is constructed. Moreover, to enable ancilla sharing, a shared cofactor that will not be required by any uncovered cubes is un-computed to recover the ancilla line to its initial state.

6. EXPERIMENTAL RESULTS

The proposed cofactor-sharing synthesis method was implemented in C++, and all experiments were done on a server machine with Intel E7-8837 processor and 64GB memory. Moreover, EXORCISM-4 [Mishchenko and Perkowski 2001] was used to initially generate an ESOP representation for each benchmark. To evaluate, we applied various experiments on circuits from quantum computing as well as MCNC benchmarks.

6.1. Quantum Benchmarks

We compared our synthesis results with the systematic method in [Markov and Saeedi 2012, Section 7.2] for LUTs that appear in Shor's algorithm. These LUTs are the four costliest modular multiplications for semiprime M values with 9 bits or less in [Markov and Saeedi 2012, Table 8]. The single-number cost model [Maslov and Saeedi 2011] is used in both methods for comparison. Synthesis results are shown in Table III. For each method a triplet $(\mathcal{T}, \mathcal{C}, \texttt{cost})$ is reported, where $\mathcal{T}$ and $\mathcal{C}$ are the number of C^2NOT (Toffoli) and CNOT gates, respectively. On average, our proposed algorithm reduces the total cost by 52%. The synthesized circuit for $M = 65$ is shown in Figure 12. As shown, a post-synthesis optimization method may further improve the results.

Additionally, since we could not find relevant synthesized results for the binary welded tree in the literature, we synthesized oracle functions in Figure 1 for black, red, green, and blue colors and applied the method in [Wille and Drechsler 2009] which is implemented in [Soeken et al. 2010] for the purpose of comparison. Synthesis results are reported in Table IV. Quantum cost and the number of ancillae are compared. As can be seen, our method leads to more compact circuits with only one ancilla as compared to the method in [Wille and Drechsler 2009].

Table III. Synthesis results for LUTs that appear in Shor's algorithm [Markov and Saeedi 2012] for semiprime M values with 9 bits or less. For each method, the number of CNOT and Toffoli gates and cost are reported as a triplet ($\mathcal{T}, \mathcal{C}, \texttt{cost}$). Our synthesis algorithm improves the results in [Markov and Saeedi 2012, Table 8] (shown as MS-2012) between 39.6% (M=253, marked with *) and 67.5% ($M = 217$, boldfaced). On average, the results in [Markov and Saeedi 2012, Table 8] are improved by 52%. Gray cells include cases where improvements are $< 45\%$. Runtimes are less than one minute in the proposed method. Both methods use at most one ancilla.

M	MS-2012	Ours	M	MS-2012	Ours	M	MS-2012	Ours	M	MS-2012	Ours
33	(49,7,252)	(16,30,110)	35	(51,7,262)	(20,31,131)	39	(44,4,224)	(16,33,113)	51	(27,4,139)	(14,12,82)
55	(47,9,244)	(18,35,125)	57	(51,6,261)	(14,30,100)	65	(41,12,217)	(15,21,96)	69	(50,7,257)	(20,48,148)
77	(55,6,281)	(24,35,155)	85	(36,2,182)	(12,19,79)	87	(56,9,289)	(20,59,159)	91	(56,6,286)	(15,45,120)
93	(50,3,253)	(15,32,107)	95	(43,9,224)	(15,54,129)	111	(51,7,262)	(14,34,104)	115	(45,11,236)	(20,41,141)
119	(57,6,291)	(15,48,123)	123	(61,6,311)	(15,58,133)	133	(50,14,264)	(10,38,88)	141	(57,8,293)	(19,43,138)
143	(49,10,255)	(20,41,141)	155	(62,11,321)	(18,52,142)	159	(52,13,273)	(18,44,134)	161	(58,11,301)	(15,51,126)
177	(48,8,248)	(17,56,141)	183	(67,11,346)	(19,66,161)	185	(61,7,312)	(17,50,135)	187	(70,9,359)	(17,72,157)
203	(63,12,327)	(19,69,164)	205	(40,3,203)	(11,31,86)	209	(60,12,312)	(17,61,146)	213	(63,13,328)	(20,80,180)
215	(62,13,323)	(17,33,118)	**217**	**(39,5,200)**	**(9,20,65)**	219	(53,9,274)	(14,46,116)	221	(60,9,309)	(15,42,117)
235	(56,16,296)	(20,55,155)	237	(62,10,320)	(20,68,168)	247	(51,11,266)	(14,58,128)	253*	(47,12,247)	(20,49,149)
259	(47,12,247)	(14,52,122)	267	(62,7,317)	(17,55,140)	287	(63,17,332)	(20,61,161)	291	(58,16,306)	(15,56,131)
295	(76,17,397)	(23,95,210)	299	(56,12,292)	(15,72,147)	301	(65,8,333)	(22,80,190)	303	(54,5,275)	(18,74,164)
305	(59,9,304)	(19,66,161)	309	(59,17,312)	(19,71,166)	319	(65,13,338)	(20,74,174)	323	(74,12,382)	(14,73,143)
327	(62,11,321)	(15,61,136)	329	(59,13,308)	(18,75,165)	335	(54,11,281)	(19,67,162)	339	(67,8,343)	(18,63,153)
341	(61,5,310)	(15,43,118)	355	(75,13,388)	(21,74,179)	365	(62,10,320)	(14,63,133)	371	(61,13,318)	(19,88,183)
377	(70,10,360)	(19,73,168)	381	(56,7,287)	(21,33,138)	391	(70,12,362)	(17,59,144)	393	(61,20,325)	(18,76,166)
395	(63,14,329)	(17,85,170)	403	(72,9,369)	(20,75,175)	407	(52,10,270)	(20,58,158)	411	(64,9,329)	(19,65,160)
413	(71,11,366)	(21,73,178)	415	(58,14,304)	(22,57,167)	417	(66,16,346)	(17,76,161)	427	(71,11,366)	(18,97,187)
437	(61,15,320)	(19,83,178)	445	(65,10,335)	(22,51,161)	447	(60,14,314)	(20,61,161)	451	(68,9,349)	(15,80,155)
453	(63,12,327)	(20,79,179)	469	(58,16,306)	(18,71,161)	471	(82,8,418)	(19,72,167)	473	(69,18,363)	(18,72,162)
481	(64,13,333)	(15,47,122)	485	(74,9,379)	(15,68,143)	493	(64,14,334)	(19,46,141)	497	(61,15,320)	(19,71,166)
501	(62,16,326)	(20,75,175)	511	(54,6,276)	(14,39,109)						

Table IV. Results for the oracles in Figure 1. For BDD-based synthesis [Wille and Drechsler 2009], shown as WD-2009, we used m CNOTs to initially copy inputs to outputs to keep inputs unchanged.

Color	(cost, #ancillae)			Color	(cost, #ancillae)		
	WD-2009	**Cofactor sharing**	**Imp.(%)**		**WD-2009**	**Cofactor sharing**	**Imp.(%)**
Blue	(339,24)	(226,1)	(33,96)	Green	(298,23)	(274,1)	(8,96)
Red	(268,21)	(256,1)	(4,95)	Black	(213,19)	(188,1)	(11,95)

6.2. MCNC Benchmarks

To evaluate the proposed method in synthesizing irreversible functions, we used the MCNC benchmarks from [Wille et al. 2008] and compared our results with methods in [Nayeem and Rice 2011] and [Lukac et al. 2011]. Since quantum cost is used in these references to calculate the synthesis cost, we also used the same cost model for reporting the cost of MCNC benchmarks.

Synthesis results for the MCNC benchmarks where at most one ancilla line is allowed to be used to construct shared cofactors are reported in Table V. Information of each benchmark circuit is given in columns 1-2 which include the name of the circuit as well as a quadruple (n, m, k, l), where n, m, k, and l denote the number of inputs, outputs, cubes, and maximal shared cofactors of each benchmark. Columns 3-5 report the results of cofactor-sharing method under various lookahead configuration parameters. In column 2, each tree node can only generate one child node, and hence just one path is traversed. As a result, in such configuration (i.e., $\Delta = 1$), the first node in the sorted $shared_cofactr_list$ is returned by Algorithm 2 regardless of the lookahead depth. This is thus a greedy algorithm which executes very fast. We set $d = 4$ and $\Delta = 5$ for column 3, whereas column 4 reports the best configuration that could produce the lowest synthesis cost in an one hour time limit. Here, G, Q, A, and T represent gate count, quantum cost, number of ancilla lines, and runtime. We compared our results with [Nayeem and Rice 2011] which is one the most recent ESOP-based synthesis methods that does not use any ancillae. On average, our simulations show 39% improvement for MCNC benchmarks which reveals the effectiveness of cofactor-sharing

Table V. Synthesis results for MCNC benchmarks. At most one ancilla is used. d and Δ indicate the depth and the maximum node degree of the lookahead tree, respectively. (n, m, k, l) denotes the number of (inputs, outputs, cubes, shared cofactors) in each benchmark. Moreover, (G, Q, A, T) is used to represent (gate count, quantum cost, number of ancilla lines, runtime). Runtimes are in seconds, unless otherwise specified. On average, results of shared-cube approach [Nayeem and Rice 2011], shown as NR-2011, are improved by 39% (compared to our best results).

Benchmarks		**Cofactor-sharing synthesis**			**NR-2011**	**Imp.**
		$d=-,\Delta=1$	$d=4,\Delta=5$	**best up to 1 hour**		(%)
Circuit	(n,m,k,l)	$(G,\mathbf{Q},A,T)$	$(G,\mathbf{Q},A,T)$	$(d,\Delta,G,\mathbf{Q},A)$	$(G,\mathbf{Q},A=0)$	
5xp1	(7,10,31,61)	(97,**544**,0,0.00)	(97,**536**,0,0.27)	(5,20,96,**519**,0)	(58,**786**)	34
9symml	(9,1,52,571)	(78,**2820**,1,0.04)	(80,**2526**,1,8.39)	(4,5,80,**2526**,1)	(52,**10943**)	77
add6	(12,7,127,401)	(217,**3864**,1,0.01)	(224,**2941**,1,3.20)	(6,5,227,**2878**,1)	-	-
adr4	(8,5,31,46)	(54,**660**,1,0.00)	(58,**518**,1,0.08)	(4,10,57,**513**,0)	-	-
alu1	(12,8,16,0)	(19,**198**,0,0.00)	(19,**198**,0,0.00)	(-,1,19,**198**,0)	-	-
alu4	(14,8,424,16046)	(653,**29115**,1,1.62)	(659,**26814**,1,6m)	(5,5,660,**26742**,1)	(454,**41127**)	35
apex4	(9,19,541,3550)	(9075,**27397**,1,0.43)	(9069,**26568**,1,2m)	(4,10,9075,**26353**,1)	(5622,**35840**)	26
apex5	(117,88,398,554)	(639,**18555**,0,0.09)	(647,**14458**,0,15.81)	(4,20,643,**11280**,0)	(601,**33830**)	67
apla	(10,12,30,128)	(107,**1063**,1,0.00)	(109,**932**,1,0.49)	(6,10,104,**875**,1)	(72,**1683**)	48
bw	(5,28,22,23)	(440,**621**,1,0.00)	(396,**581**,1,0.05)	(6,10,375,**561**,1)	(287,**637**)	12
C17	(5,2,6,4)	(10,**54**,0,0.00)	(10,**54**,0,0.00)	(-,1,10,**54**,0)	-	-
C7552	(5,16,16,15)	(135,**281**,1,0.00)	(122,**247**,1,0.01)	(4,10,115,**236**,1)	(89,**399**)	41
clip	(9,5,64,340)	(166,**2156**,1,0.02)	(163,**2118**,1,2.61)	(5,10,167,**2000**,1)	(78,**3824**)	48
cm150a	(21,1,17,64)	(33,**625**,1,0.00)	(33,**625**,1,0.30)	(-,1,33,**625**,1)	-	-
cm151a	(19,9,23,13)	(26,**456**,1,0.00)	(26,**281**,1,0.00)	(3,5,26,**281**,1)	-	-
cm152a	(11,1,8,12)	(16,**144**,1,0.00)	(16,**144**,1,0.00)	(-,1,16,**144**,1)	-	-
cm163a	(16,13,19,16)	(38,**334**,0,0.00)	(36,**299**,0,0.02)	(3,5,36,**299**,0)	-	-
cm42a	(4,10,11,7)	(52,**119**,0,0.00)	(52,**119**,0,0.00)	(-,1,52,**119**,0)	(42,**161**)	26
cmb	(16,4,4,1)	(9,**451**,0,0.00)	(9,**451**,0,0.00)	(-,1,9,**451**,0)	-	-
cordic	(23,2,776,6656)	(2324,**82975**,1,0.37)	(2331,**64928**,1,5.96)	(4,10,2337,**51572**,1)	(777,**187620**)	73
cu	(14,11,16,20)	(36,**363**,0,0.00)	(35,**357**,0,0.00)	(2,5,35,**357**,0)	(28,**781**)	54
dc1	(4,7,9,8)	(45,**113**,1,0.00)	(46,**110**,1,0.00)	(4,10,42,**106**,1)	(31,**127**)	17
dc2	(8,7,32,88)	(93,**737**,0,0.00)	(95,**697**,1,0.24)	(7,10,96,**675**,1)	(51,**1084**)	38
decod	(5,16,16,15)	(135,**281**,1,0.00)	(122,**247**,1,0.01)	(4,10,115,**236**,1)	(89,**399**)	41
dist	(8,5,68,430)	(185,**2294**,1,0.01)	(190,**2164**,1,2.26)	(6,10,198,**1940**,1)	(94,**3700**)	48
dk17	(10,11,21,60)	(49,**643**,0,0.00)	(50,**598**,0,0.16)	(5,10,49,**589**,0)	(34,**1014**)	42
ex1010	(10,10,648,16217)	(3549,**40093**,1,2.29)	(3560,**39406**,1,9m)	(5,5,3563,**37738**,1)	(1675,**52788**)	29
ex5p	(8,63,72,215)	(1087,**2583**,1,0.01)	(1021,**2273**,1,2.22)	(5,10,969,**2208**,1)	(646,**3547**)	38
f2	(4,4,8,7)	(21 ,**97**,1,0.00)	(20,**84**,1,0.00)	(3,5,20,**84**,1)	(14,**112**)	25
f51m	(14,8,287,5502)	(430 ,**20001**,1,0.28)	(440,**16669**,1,1m)	(6,5,443,**15838**,1)	(327,**28382**)	44
frg1	(28,3,115,975)	(133 ,**8937**,1,0.06)	(142,**7808**,1,9.30)	(5,10,144,**7630**,1)	-	-
frg2	(143,139,1116,1923)	(2679,**62870**,0,0.85)	(2687,**60352**,1,27.87)	(3,20,2687,**60178**,1)	(1389,**112008**)	46
ham7	(7,7,11,1)	(49 ,**65**,0,0.00)	(49,**65**,0,0.00)	(-,1,49,**65**,0)	(37,**67**)	3
hwb8	(8,8,192,927)	(1078,**5743**,1,0.09)	(1079,**5777**,1,16.03)	(6,5,1086,**5499**,1)	(480,**8195**)	33
in0	(15,11,92,528)	(388 ,**4706**,1,0.02)	(400,**4274**,1,1.31)	(5,20,402,**4136**,1)	(245,**7949**)	48
inc	(7,9,27,72)	(128 ,**622**,0,0.00)	(122,**548**,1,0.21)	(5,20,124,**546**,1)	(75,**892**)	39
max46	(9,1,41,466)	(67 ,**2310**,1,0.02)	(65,**2202**,1,4.25)	(5,5,67,**2144**,1)	-	-
misex1	(8,7,12,15)	(55 ,**225**,0,0.00)	(54,**220**,1,0.01)	(4,20,53,**211**,1)	(42,**332**)	36
misex3	(14,14,507,14749)	(1915,**31393**,1,1.60)	(1909,**31050**,1,6m)	(5,5,1913,**29314**,1)	(854,**49076**)	40
misex3c	(14,14,512,15111)	(1963,**30947**,1,1.83)	(1963,**31793**,1,7m)	(2,10,1962,**30771**,1)	(822,**49720**)	38
mlp4	(8,8,60,232)	(132 ,**1748**,1,0.01)	(136,**1436**,1,0.97)	(7,10,138,**1280**,0)	(80,**2496**)	49
mux	(21,1,16,64)	(32 ,**624**,1,0.00)	(32,**624**,1,0.29)	(-,1,32,**624**,)	-	-
pdc	(16,40,254,4627)	(1335,**21635**,0,0.19)	(1341,**18765**,0,25.23)	(5,10,1347,**16722**,0)	(649,**30962**)	46
pm1	(4,10,11,5)	(71 ,**126**,0,0.00)	(66,**117**,0,0.00)	(3,5,66,**117**,0)	-	-
root	(8,5,35,170)	(105 ,**1230**,1,0.01)	(107,**1195**,1,0.76)	(5,20,107,**1092**,1)	(48,**1811**)	40
ryy6	(16,1,40,52)	(47 ,**1857**,1,0.00)	(51,**1456**,1,0.26)	(6,5,52,**1420**,1)	-	-
sao2	(10,4,28,139)	(96 ,**1645**,1,0.00)	(97,**1439**,1,0.30)	(7,10,100,**1338**,1)	(41,**3767**)	64
seq	(41,35,246,1768)	(3374,**25917**,0,0.11)	(3358,**21485**,1,4.82)	(4,20,3343,**18873**,1)	(1287,**33991**)	44
sqr6	(6,12,33,60)	(80 ,**517**,1,0.00)	(83,**447**,1,0.17)	(5,10,82,**442**,1)	(54,**583**)	24
sqrt8	(8,4,17,21)	(39 ,**296**,1,0.00)	(41,**293**,1,0.01)	(4,10,40,**288**,0)	-	-
sym9	(9,1,52,569)	(78 ,**2626**,1,0.03)	(78,**2536**,1,7.53)	(4,5,78,**2536**,1)	-	-
sym10	(10,1,78,1195)	(106 ,**4459**,1,0.08)	(108,**4300**,1,17.70)	(5,10,108,**4040**,1)	-	-
t481	(16,1,13,8)	(19 ,**142**,1,0.00)	(19,**142**,1,0.01)	(-,1,19,**142**,1)	-	-
tial	(14,8,428,17060)	(632 ,**30271**,1,1.26)	(637,**29106**,1,4m)	(4,10,638,**28475**,1)	-	-
urf3	(10,10,752,12662)	(3730,**40102**,1,1.51)	(3735,**37822**,1,6m)	(5,5,3739,**37663**,1)	(1501,**53157**)	29
wim	(4,7,10,8)	(34 ,**97**,0,0.00)	(33,**96**,0,0.00)	(2,5,33,**96**,0)	(23,**139**)	31
x2	(10,7,15,23)	(41 ,**325**,0,0.00)	(40,**304**,0,0.03)	(4,5,40,**304**,0)	-	-
z4ml	(7,4,29,36)	(62 ,**402**,1,0.00)	(62,**402**,1,0.10)	(5,10,63,**397**,1)	(34,**489**)	19

Table VI. Synthesis results for MCNC benchmarks. Extra ancilla lines may be used to construct large cubes. (n, m, k, l) denotes the number of (inputs, outputs, cubes, shared cofactors) in each benchmark. G, Q and A are used to represent gate count, quantum cost, and number of ancilla lines. θ_c indicates the threshold value of detecting a large cube. On average, while costs of cube-reordering approach [Lukac et al. 2011], shown as LKPP-2011, are degraded by 13%, ancilla count is improved by 71%.

Benchmarks		**Cofactor-sharing synthesis**		**LKPP-2011**	**Imp.**
		ancilla budget = 5	**no ancilla budget**		**(%)**
Circuit	(n, m, k, l)	$(\theta_c, G, \mathbf{Q}, A)$	$(\theta_c, G, \mathbf{Q}, A)$	$(G, \mathbf{Q}, A)$	(Q, A)
5xp1	(7,10,31,61)	(1,97,**506**,1)	(1,97,**506**,1)	-	-
9symml	(9,1,52,571)	(4,80,**2526**,1)	(4,80,**2526**,1)	(571,**2831**,11)	(11,91)
add6	(12,7,127,401)	(4,227,**2878**,1)	(4,227,**2878**,1)	(2246,**11026**,11)	(74,91)
adr4	(8,5,31,46)	(3,57,**513**,0)	(3,57,**513**,0)	(332,**1492**,11)	(66,100)
alu1	(12,8,16,0)	(1,19,**198**,0)	(1,19,**198**,0)	(26,**102**,11)	(-94,100)
alu4	(14,8,424,16046)	(3,800,**24303**,5)	(3,1006,**22086**,23)	-	-
apex4	(9,19,541,3550)	(1,9289,**25198**,5)	(2,9559,**24270**,20)	(2388,**11912**,38)	(-104,47)
apex5	(117,88,398,554)	(8,661,**10839**,3)	(8,661,**10839**,3)	(4479,**22115**,118)	(51,97)
apla	(10,12,30,128)	(2,106,**841**,3)	(2,106,**841**,3)	(106,**530**,10)	(-59,70)
bw	(5,28,22,23)	(2,375,**561**,1)	(2,375,**561**,1)	-	-
C17	(5,2,6,4)	(1,10,**54**,0)	(1,10,**54**,0)	(6,**22**,0)	(-145,0)
C7552	(5,16,16,15)	(2,115,**236**,1)	(2,115,**236**,1)	(56,**280**,3)	(16,67)
clip	(9,5,64,340)	(3,174,**1976**,4)	(2,184,**1962**,8)	(833,**3980**,8)	(51,0)
cm150a	(21,1,17,64)	(1,33,**625**,1)	(1,33,**625**,1)	(110,**546**,20)	(-14,95)
cm151a	(19,9,23,13)	(3,26,**281**,1)	(3,26,**281**,1)	(78,**390**,18)	(28,94)
cm152a	(11,1,8,12)	(1,16,**144**,1)	(1,16,**144**,1)	(30,**150**,10)	(4,90)
cm163a	(16,13,19,16)	(1,36,**299**,0)	(1,36,**299**,0)	(56,**128**,15)	(-134,100)
cm42a	(4,10,11,7)	(1,52,**119**,0)	(1,52,**119**,0)	-	-
cmb	(16,4,4,1)	(1,9,**451**,0)	(1,9,**451**,0)	(71,**243**,15)	(-86,100)
cordic	(23,2,776,6656)	(5,2718,**40088**,4)	(5,2718,**40088**,4)	-	-
cu	(14,11,16,20)	(4,35,**357**,0)	(4,35,**357**,0)	-	-
dc1	(4,7,9,8)	(1,42,**106**,1)	(1,42,**106**,1)	-	-
dc2	(8,7,32,88)	(4,96,**675**,1)	(4,96,**675**,1)	-	-
decod	(5,16,16,15)	(2,115,**236**,1)	(2,115,**236**,1)	-	-
dist	(8,5,68,430)	(2,212,**1906**,5)	(2,216,**1882**,8)	-	-
dk17	(10,11,21,60)	(1,49,**589**,0)	(1,49,**589**,0)	-	-
ex1010	(10,10,648,16217)	(1,3767,**34643**,5)	(2,4113,**31588**,26)	-	-
ex5p	(8,63,72,215)	(3,973,**2197**,2)	(3,973,**2197**,2)	-	-
f2	(4,4,8,7)	(1,20,**84**,1)	(1,20,**84**,1)	-	-
f51m	(14,8,287,5502)	(1,563,**14898**,5)	(3,709,**14669**,19)	-	-
frg1	(28,3,115,975)	(7,161,**7410**,5)	(1,220,**7007**,13)	(582,**2898**,27)	(-142,52)
frg2	(143,139,1116,1923)	(7,3037,**55424**,5)	(7,3046,**55532**,6)	(19361,**95469**,142)	(42,96)
ham7	(7,7,11,1)	(1,49,**65**,0)	(1,49,**65**,0)	-	-
hwb8	(8,8,192,927)	(1,1148,**5175**,5)	(2,1199,**4827**,18)	-	-
in0	(15,11,92,528)	(8,402,**4136**,1)	(8,402,**4136**,1)	-	-
inc	(7,9,27,72)	(3,124,**546**,1)	(3,124,**546**,1)	-	-
max46	(9,1,41,466)	(1,68,**2131**,2)	(1,68,**2131**,2)	(419,**2095**,8)	(-2,75)
misex1	(8,7,12,15)	(1,54,**211**,2)	(1,54,**211**,2)	(42,**170**,7)	(-24,71)
misex3	(14,14,507,14749)	(3,2012,**28490**,5)	(3,2094,**27716**,15)	(8394,**40846**,13)	(32,-15)
misex3c	(14,14,512,15111)	(4,2128,**30327**,5)	(2,2299,**29689**,17)	-	-
mlp4	(8,8,60,232)	(4,138,**1280**,0)	(4,138,**1280**,0)	-	-
mux	(21,1,16,64)	(1,32,**624**,1)	(1,32,**624**,1)	(289,**1433**,20)	(56,95)
pdc	(16,40,254,4627)	(9,1347,**16722**,0)	(9,1347,**16722**,0)	-	-
pm1	(4,10,11,5)	(1,66,**117**,0)	(1,66,**117**,0)	(40,**40**,3)	(-193,100)
root	(8,5,35,170)	(1,117,**1054**,5)	(1,118,**1054**,6)	-	-
ryy6	(16,1,40,52)	(2,69,**1363**,5)	(2,73,**1315**,6)	(281,**1405**,15)	(6,60)
sao2	(10,4,28,139)	(3,100,**1338**,1)	(3,100,**1338**,1)	-	-
seq	(41,35,246,1768)	(13,3343,18873,1)	(13,3343,**18873**,1)	(11470,**56422**,40)	(67,98)
sqr6	(6,12,33,60)	(3,82,**442**,1)	(3,82,**442**,1)	-	-
sqrt8	(8,4,17,21)	(1,42,**263**,1)	(1,42,**263**,1)	(94,**462**,7)	(43,86)
sym9	(9,1,52,569)	(3,78,**2536**,1)	(3,78,**2536**,1)	(649,**3133**,8)	(19,88)
sym10	(10,1,78,1195)	(1,117,**3928**,5)	(1,119,**3928**,6)	(3591,**17955**,9)	(78,33)
t481	(16,1,13,8)	(1,19,**142**,1)	(1,19,**142**,1)	(2792,**13924**,15)	(99,93)
tial	(14,8,428,17060)	(1,794,**26172**,5)	(1,1020,**24626**,21)	(5087,**25211**,14)	(2,-50)
urf3	(10,10,752,12662)	(2,4115,**34574**,5)	(3,4538,**32188**,26)	-	-
wim	(4,7,10,8)	(1,33,**96**,0)	(1,33,**96**,0)	-	-
x2	(10,7,15,23)	(1,40,**304**,0)	(1,40,**304**,0)	(41,**129**,9)	(-136,100)
z4ml	(7,4,29,36)	(2,63,**397**,1)	(2,63,**397**,1)	-	-

in reversible logic synthesis.[4] Finally, we provided the cofactor-sharing synthesis with more ancillae to evaluate its ability in improving the cost of large cubes. Results are reported in Table VI. In one case (column 3), we restricted the number of ancillae to 5, while in the other case (column 4), there is no limitation on the number of available ancilla lines. For each synthesis result, we also report the threshold value (shown as θ_c) that is used to identify large cubes. We compared our results with [Lukac et al. 2011], a synthesis algorithm based on cube-reordering which extensively uses ancillae. On average, though synthesis costs are degraded by 13%, ancilla count, which is a very limited resource in quantum technologies, is improved by 71%.

7. CONCLUSIONS

We addressed the problem of synthesizing a given function on a set of ancillea by reversible gates. Our algorithm is based on extensive sharing of cofactors to reuse shared cubes without applying additional reversible gates. In particular, the proposed approach tries different cofactors at each step with a lookahead strategy. To construct cofactors on a limited number of qubits, the algorithm uses cofactor construction with un-computation. Our experiments showed the proposed method can significantly (52% on average) improve the synthesis cost of a recent method for those LUTs that appear in Shor's factoring algorithm. The results of applying the proposed method on the MCNC benchmarks show a considerable improvement in cost (39% on average) as compared with a recent ESOP-based method. The proposed approach can be limited to use a restricted set of ancillea for cost reduction. We also showed that cofactor sharing can improve oracle of a binary welded tree.

REFERENCES

D. Bacon and W. van Dam. 2010. Recent Progress in Quantum Algorithms. *Commun. ACM* 53 (Feb. 2010), 84–93. Issue 2.

R. Brayton, A. L. Sangiovanni-Vincentelli, C. T. McMullen, and G. D. Hachtel. 1984. *Logic Minimization Algorithms for VLSI Synthesis*. Kluwer Academic Publishers.

C.-F. Chiang, D. Nagaj, and P. Wocjan. 2010. Efficient Circuits for Quantum Qalks. *Quantum Info. Comput.* 10, 5 (May 2010), 420–434.

A. M. Childs, R. Cleve, E. Deotto, E. Farhi, S. Gutmann, and D. A. Spielman. 2003. Exponential Algorithmic Speedup by a Quantum Walk. In *Thirty-fifth Annual ACM Symposium on Theory of Computing*. 59–68.

O. Coudert. 1994. Two-level Logic Minimization: An Overview. *Integr. VLSI J.* 17, 2 (Oct. 1994), 97–140.

K. Fazel, M.A. Thornton, and J.E. Rice. 2007. ESOP-based Toffoli Gate Cascade Generation. In *Communications, Computers and Signal Processing, IEEE Pacific Rim Conference on*. 206 –209.

M. Lukac, M. Kameyama, M. Perkowski, and P. Kerntopf. 2011. Decomposition of Reversible Logic Function Based on Cube-Reordering. In *Facta Universitatis - series: Electronics and Energetics*, Vol. 24. 403–422.

I. L. Markov and M. Saeedi. 2012. Constant-optimized Quantum Circuits for Modular Multiplication and Exponentiation. *Quantum Info. Comput.* 12, 5-6 (May 2012), 361–394.

D. Maslov and M. Saeedi. 2011. Reversible Circuit Optimization via Leaving the Boolean Domain. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* 30, 6 (Jun. 2011), 806 – 816.

A. Mishchenko and M. Perkowski. 2001. Fast Heuristic Minimization of Exclusive Sum-of-Products. In *Reed-Muller Workshop*.

N. M. Nayeem and J. E. Rice. 2011. A Shared-cube Approach to ESOP-based Synthesis of Reversible Logic. In *Facta Universitatis - series: Electronics and Energetics*, Vol. 24. 385–402.

[4]Exploring various orderings of shared cofactors and dependent cubes by an exhaustive approach can improve synthesis costs. Our experiment for small circuits with ≤ 10 dependent cubes for each shared cofactor, circuits with boldfaced names in Table V, showed that synthesis costs can slightly be improved. In particular, the best improvement was obtained for `pm1` benchmark — cost was reduced from 117 to 113. Optimization by an exhaustive approach for large circuits with many different dependent cubes and shared cofactors may result in improved costs but can be very time-consuming.

M. A. Nielsen and I. L. Chuang. 2000. *Quantum Computation and Quantum Information*. Cambridge University Press.

M. Plesch and C. Brukner. 2011. Quantum-state preparation with universal gate decompositions. *Phys. Rev. A* 83 (Mar 2011), 032302. Issue 3.

M. Saeedi and I. L. Markov. 2013. Synthesis and Optimization of Reversible Circuits A Survey. *Comput. Surveys* 45, 2 (March 2013), 21:1–21:34.

M. Saeedi, M. Saheb Zamani, M. Sedighi, and Z. Sasanian. 2010. Reversible Circuit Synthesis using a Cycle-based Approach. *J. Emerg. Technol. Comput. Sys.* 6, 4 (Dec. 2010), 13:1–13:26.

G. Schaller and R. Schützhold. 2010. The role of symmetries in adiabatic quantum algorithms. *Quantum Info. Comput.* 10, 1 (Jan. 2010), 109–140.

A. Shafaei, M. Saeedi, and M. Pedram. 2013. Reversible Logic Synthesis of k-input, m-output Lookup Tables. In *Design, Automation and Test in Europe*. 1235–1240.

M. Soeken, S. Frehse, R. Wille, and R. Drechsler. 2010. RevKit: A Toolkit for Reversible Circuit Design. *Workshop on Reversible Computation* (2010).

L. M. K. Vandersypen, M. Steffen, G. Breyta, C. S. Yannoni, R. Cleve, and I. L. Chuang. 2000. Experimental Realization of an Order-Finding Algorithm with an NMR Quantum Computer. *Phys. Rev. Lett.* 85 (Dec. 2000), 5452–5455. Issue 25.

R. Wille and R. Drechsler. 2009. BDD-based Synthesis of Reversible Logic for Large Functions. In *Design Automation Conference*. 270–275.

R. Wille, D. Große, L. Teuber, G. W. Dueck, and R. Drechsler. 2008. RevLib: An Online Resource for Reversible Functions and Reversible Circuits. In *Int'l Symp. on Multi-Valued Logic*. 220–225.

N. Xu, J. Zhu, D. Lu, X. Zhou, X. Peng, and J. Du. 2012. Quantum Factorization of 143 on a Dipolar-Coupling Nuclear Magnetic Resonance System. *Phys. Rev. Lett.* 108 (Mar. 2012), 130501. Issue 13.