

Effect of BDD Optimization on Synthesis of Reversible and Quantum Logic

Robert Wille¹ Rolf Drechsler²

*Institute of Computer Science
University of Bremen
Bremen, Germany*

Abstract

Synthesis of reversible and quantum logic has become an intensely studied topic in the last years. However, most synthesis methods are limited, since they rely on a truth table representation of the function to be synthesized. BDD-based synthesis offers an alternative. Here, reversible or quantum circuits are derived from a function given as *Binary Decision Diagram* (BDD) by substituting all nodes of the BDD with a cascade of Toffoli or elementary quantum gates, respectively. As a result, the application of the approach is not limited by the truth table of the function but by the (quite more efficient) BDD representation. Furthermore, many optimization techniques for BDDs exist which can be exploited. In this work, we evaluate the effect of three optimization methods for BDDs (namely shared nodes, complement edges, and advanced orderings) on the resulting reversible and quantum circuits. We describe in detail the adjustments, which have to be done to support these optimizations for synthesis, and discuss possible improvements and drawbacks. In a case study, the effects are experimentally evaluated. The results showed, that applying these optimization techniques leads to significant smaller circuits (with respect to number of gates and lines) in most of the cases.

Keywords: Synthesis, Reversible Circuits, Quantum Circuits, Binary Decision Diagrams

1 Introduction

Reversible and quantum logic [10,1,20] has applications in domains like low-power design [10], quantum computing [15], optical computing [4], DNA computing [1], and nanotechnologies [13]. Since synthesis of reversible and quantum circuits significantly differs from traditional design (e.g. fan-out and feedback are not allowed), it has become an intensely studied research area in the recent years.

However, many synthesis approaches are limited. Exact (see e.g. [7,23]) as well as heuristic (see e.g. [18,11,6,12]) methods have been proposed. But both are applicable only for relatively small functions. Exact approaches reach their limits with

¹ Email: rwille@informatik.uni-bremen.de

² Email: drechsler@uni-bremen.de

functions containing more than 6 variables [23] while heuristic methods are able to synthesize functions with at most 30 variables [6]. Moreover, often a significant amount of run-time is needed to achieve these results.

These limitations are mainly caused by the underlying techniques. The existing synthesis approaches often rely on truth tables (or similar descriptions like permutations) of the function to be synthesized (e.g. in [18,14]). But even if more compact data structures like BDDs [9], positive-polarity Reed-Muller expansion [6], or Reed-Muller spectra [12] are used, the same limitations can be observed since all of them apply similar strategies (namely selecting reversible gates so that the chosen function representation becomes the identity).

As an alternative, in [21] a new synthesis approach has been introduced that can cope with significantly larger functions. Here, reversible or quantum circuits are derived from a function given as BDD [3] by substituting all nodes of the BDD with a cascade of Toffoli or elementary quantum gates, respectively. As a result, the synthesis approach is not limited by the truth table of the function but by the (quite more efficient) BDD representation. However, since for BDDs many optimization techniques have been developed (e.g. *shared nodes* [3], *complement edges* [2], and reordering strategies like *sifting* [17]) it seems obvious to exploit these optimizations for the synthesis of reversible and quantum logic as well. But this requires new methods to derive circuits from the BDD.

In this work, we describe an improved BDD-based synthesis approach that supports shared nodes, complement edges, and different orderings for BDD-based synthesis of reversible and quantum logic and discuss possible improvements and drawbacks. In a case study, we evaluate the effect of these optimization methods on the resulting circuit sizes. It turned out, that applying these optimization techniques leads to significant smaller circuits in most of the cases.

The remainder of the paper is structured as follows: Section 2 provides the basics of reversible and quantum logic as well as of BDDs. Afterwards, in Section 3 the synthesis approach as proposed in [21] is briefly reviewed. Section 4 describes the new BDD-based synthesis approach that supports shared nodes, complement edges, and reordering for BDD-based synthesis of reversible and quantum logic. Finally, in Section 5 the effect of these optimization techniques on the resulting circuits is experimentally evaluated while the paper is concluded in Section 6.

2 Preliminaries

To keep the paper self-contained this section briefly reviews the basic concepts of reversible and quantum logic. We also describe the basics of BDDs which are used as the underlying data structure by the synthesis approach.

2.1 Reversible Logic

A logic function is reversible if it maps each input assignment to a unique output assignment. Such a function must have the same number of input and output variables $X := \{x_1, \dots, x_n\}$. Since fanout and feedback are not allowed in reversible

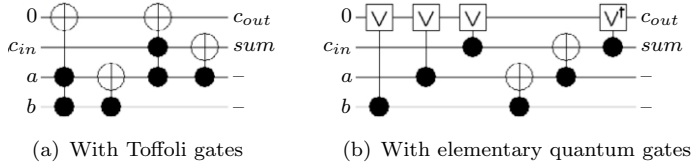


Fig. 1. Two circuits realizing a full adder

logic, a circuit realizing a reversible function is a cascade of reversible gates. A *reversible gate* has the form $g(C, T)$, where $C = \{x_{i_1}, \dots, x_{i_k}\} \subset X$ is the set of control lines and $T = \{x_{j_1}, \dots, x_{j_l}\} \subset X$ with $C \cap T = \emptyset$ is the set of target lines. C may be empty. The gate operation is applied to the target lines iff all control lines meet the required control conditions. Control lines and unconnected lines always pass through the gate unaltered.

In the literature, several types of reversible gates have been introduced. Besides the Fredkin gate [5] and the Peres gate [16]), (*multiple controlled*) *Toffoli* gates [20] are widely used. Each Toffoli gate has one target line x_j , which is inverted iff all control lines are assigned to 1. That is, a multiple controlled Toffoli gate maps $(x_1, \dots, x_j, \dots, x_n)$ to $(x_1, \dots, x_{i_1}x_{i_2} \cdots x_{i_k} \oplus x_j, \dots, x_n)$.

Quantum circuits realize functions with the help of *elementary quantum gates*. Quantum circuits are inherently reversible and manipulate qubits rather than pure logic values. The state of a qubit for two pure logic states can be expressed as $|\Psi\rangle = \alpha|0\rangle + \beta|1\rangle$, where $|0\rangle$ and $|1\rangle$ denote pure logic states 0 and 1, respectively, and α and β are complex numbers such that $|\alpha|^2 + |\beta|^2 = 1$. The most frequently occurring elementary quantum gates are the NOT gate (a single qubit is inverted), the controlled-NOT (CNOT) gate (the target qubit is inverted if the single control qubit is 1), the controlled-V gate (also known as a square root of NOT, since two consecutive V operations are equivalent to an inversion), and the controlled-V+ gate (which performs the inverse operation of the V gate and thus is also a square root of NOT).

Example 2.1 Figure 1(a) shows a Toffoli gate realization of a full adder. This circuit has four inputs (the constant input 0, the carry-in c_{in} , as well as the summands a and b), four outputs (the carry-out c_{out} and the *sum* as well as two garbage outputs), and consists of four Toffoli gates. Thereby, the control lines of each Toffoli gate are denoted by \bullet while the target lines are denoted by \oplus . A circuit realizing the same function by elementary quantum gates is depicted in Figure 1(b). This circuit has the same inputs and outputs but consists of six gates in total. The notation is similar to a Toffoli circuit, except that the target lines are denoted with respect to the particular gate type. More precisely, a V-box is used to denote a controlled-V gate and a V+-box is used to denote a controlled-V+ gate. The notations for NOT and CNOT gates are equal to the notation for Toffoli gates.

2.2 Binary Decision Diagrams

A Boolean function $f : \mathbb{B}^n \rightarrow \mathbb{B}$ can be represented by a *Binary Decision Diagram* (BDD) [3]. A BDD is a directed acyclic graph $G = (V, E)$ where a Shannon decomposition

$$f = \bar{x}_i f_{x_i=0} + x_i f_{x_i=1} \quad (1 \leq i \leq n)$$

is carried out in each node $v \in V$. The functions $f_{x_i=0}$ and $f_{x_i=1}$ are the *cofactors* of f . In the following the node representing $f_{x_i=0}$ ($f_{x_i=1}$) is denoted by *low*(v) (*high*(v)) while x_i is called the *select variable*. A BDD is called free if each variable is encountered at most once on each path from the root to a terminal node. A BDD is called ordered if in addition all variables are encountered in the same order on all such paths. In the following, ordered binary decision diagrams are called BDD for brevity. The *size* k of a BDD is defined by the number of nodes.

In the past, several techniques to optimize the size of BDDs have been developed. In particular *shared nodes* [3] allow significant reductions. That is, if a node v has more than one predecessor. In particular, functions $f : \mathbb{B}^n \rightarrow \mathbb{B}^m$ (i.e. functions with more than one output) can be represented more compactly using shared nodes. Further reduction can be achieved if *complement edges* [2] are applied. This enables the representation of a function as well as of its negation by a single node only. Furthermore, the size of a BDD significantly depends on the chosen ordering of its input variables [3].

3 BDD-based Synthesis

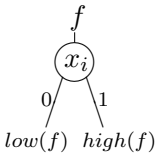
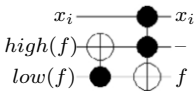
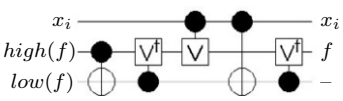
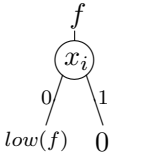
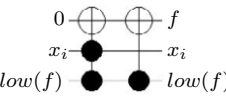
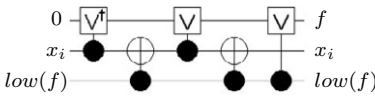
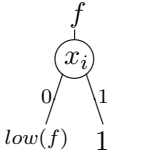
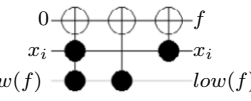
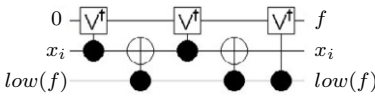
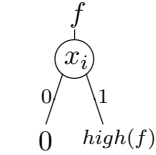
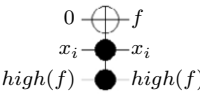
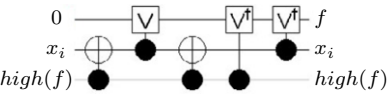
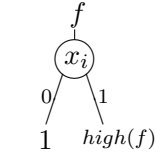
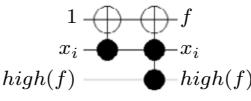
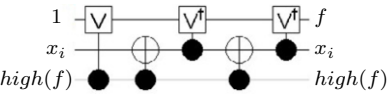
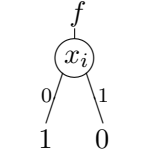


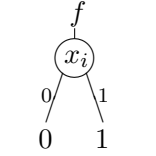


In this section, we briefly review BDD-based synthesis of reversible and quantum logic as introduced in [21]. This provides the basis for the rest of this paper, where the application of BDD optimization techniques to the synthesis approach is discussed in detail.

The aim of each synthesis approach is to determine a circuit realization for a given Boolean function. It is well known, that Boolean functions can be efficiently represented by BDDs [3]. Having a BDD $G = (V, E)$, a reversible network can be derived by traversing the BDD and substituting each node $v \in V$ with a cascade of reversible gates. The respective cascade of gates depends on the successors of the node v . Table 1 provides the cascades of Toffoli and elementary quantum gates, respectively, for all possible scenarios of a BDD node.

Note that an additional (constant) line is necessary if one of the edges *low*(v) or *high*(v) leads to a terminal node. This is because of the reversibility which has to be ensured when synthesizing reversible logic. As an example consider a node v with *high*(v) = 0 (second row of Table 1). Without loss of generality, the first three lines of the corresponding truth table can be embedded with respect to reversibility as depicted in Table 2(a). However, since f is 0 in the last line, no reversible embedding for the whole function is possible. Thus, an additional line is required to make the respective substitution reversible (see Table 2(b))³.

³ Due to the same reason it is also not possible to preserve the values for *low*(v) or *high*(v), respectively,

Table 1
Substitution of BDD nodes to reversible/quantum circuit

BDD	Toffoli Circuit	Quantum circuit
		
		
		
		
		
		
		

Based on these substitutions, a method for synthesizing Boolean functions in reversible or quantum logic can be formulated: First, a BDD for function f to be synthesized is created. This can be done efficiently using state-of-the-art BDD packages (e.g. CUDD [19]). Next, the resulting BDD $G = (V, E)$ is traversed by a depth-first search. For each node $v \in V$, cascades as depicted in Table 1 are added

in the substitution depicted in the first row of Table 1.

Table 2
(Partial) Truth tables for node v with $high(v) = 0$

(a) w/o add. line				(b) with additional line					
x_i	$low(f)$	f	$-$	0	x_i	$low(f)$	f	x_i	$low(f)$
0	0	0	0	0	0	0	0	0	0
0	1	1	1	0	0	1	1	0	1
1	0	0	1	0	1	0	0	1	0
1	1	0	?	0	1	1	0	1	1

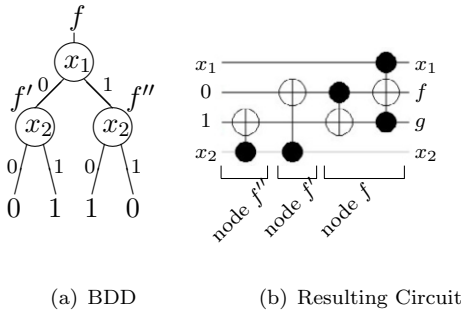


Fig. 2. BDD and Toffoli circuit for $f = x_1 \oplus x_2$

to the circuit.

Example 3.1 Consider the BDD in Figure 2(a). Applying the substitutions given in Table 1 to each node of the BDD, the Toffoli network depicted in Figure 2(b) results.

As a result, circuits are synthesized which realize the given function f . Since, each node of the BDD is only substituted by a cascade of gates, the proposed method has a linear worst case run-time and memory complexity with respect to the number of nodes in the BDD.

4 Exploiting BDD Optimization

Current state-of-the-art BDD packages (e.g. CUDD [19]) exploit several optimization techniques to build BDDs of small size. In this section, we describe how these techniques can be applied to the proposed BDD-based synthesis as well. The effect of these optimizations on the resulting reversible or quantum circuits is considered in the next section.

4.1 Shared Nodes

If a node v has more than one predecessor, then v is called a *shared node*. The application of shared nodes is common for nearly all BDD packages. Shared nodes can be used to represent a sub-formula more than once without the need to rebuild the whole sub-graph. In particular, functions $f : \mathbb{B}^n \rightarrow \mathbb{B}^m$ (i.e. functions with more than one output) can be represented more compactly using shared nodes.

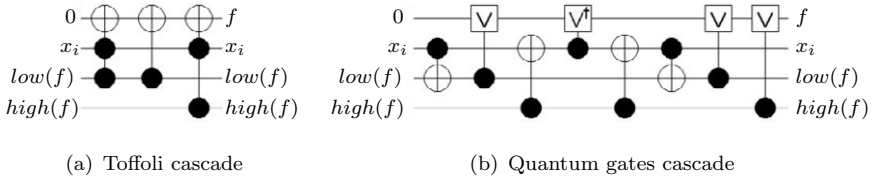


Fig. 3. Substitution for shared nodes without terminals as successors

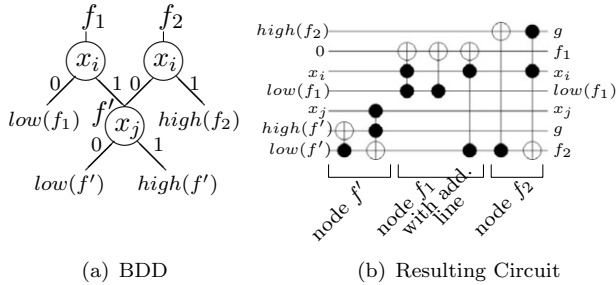


Fig. 4. Toffoli circuit for shared BDD

However, to use shared nodes, the output value of a respective node has to be preserved until it is not needed any longer. To ensure this, additional circuit lines with constant inputs are required. Considering the substitutions depicted in Table 1, this is the case for all nodes v where one of the edges $low(v)$ or $high(v)$ leads to a terminal node. Here, all values of the inputs (in particular of $high(v)$ or $low(v)$, respectively) are preserved. But as already mentioned above, this is not possible for the general case (first row of Table 1). Here, at least one value (namely the value from the select variable) is preserved. Thus, a modified substitution for shared nodes without terminals as successors is required. Figure 3(a) and Figure 3(b) show one possible substitution for reversible and one for quantum circuits, respectively.

Besides the additional (constant) circuit line, this requires one additional gate in comparison to the substitution of Table 1. In contrast, using this substitution no cascade for nodes representing the identity of their select variable is necessary any longer. In this case, the node can be represented by the same circuit line as the input itself. This is now possible since the values of circuit lines representing a node can be preserved.

Example 4.1 In Figure 4(a) a partial BDD including a shared node f' is shown. Since the value of node f' is used twice (by nodes f_1 and f_2), an additional line (line 2 in Figure 4(b)) and the cascade of gates as depicted in Figure 3(a) is applied to substitute node f_1 . Then, the value of f' is still available such that the substitution of node f_2 can be applied. The resulting circuit is given in Figure 4(b).

Table 3
Substitution of BDD nodes with complement edge to reversible/quantum circuit

BDD	Toffoli Circuit	Quantum circuit

4.2 Complement Edges

Further reductions in BDD sizes can be achieved if *complement edges* [2] are applied. In particular, this allows to represent a function as well as its negation by a single node only. If a complement edge is applied, the output value of its connected node becomes inverted. To support complement edges in our synthesis approach, new substitutions have to be determined, that take the inversion by complement edges into account.

Table 3 shows the resulting cascades used in our synthesis approach. Note, that complements have to be considered only at the *low* edges of the nodes. Nodes with complements at the *high*-edge can easily be replaced by a respective node with a complement at the *low* edge.

In some cases, this leads to larger cascades in comparison to the substitution without complement edges (in particular for the Toffoli cascades). How far this can be compensated by the possible BDD reductions (or not) is discussed in detail in Section 5.

4.3 Ordering of BDDs

It has been shown that the order of the variables has a high impact on the size of the resulting BDD [3]. In the past several approaches have been proposed to achieve good orderings (e.g. sifting [17]) or to determine exact results [8]. All these techniques can be directly applied to our synthesis approach and need no further adjustments of the already introduced substitutions.

Using these optimization techniques (i.e. shared nodes, complement edges, and reordering), in the next section it is considered how they influence the resulting Toffoli or quantum circuits, respectively.

5 Experimental Study

In this section we investigate the effect of the respective optimization techniques for BDD minimization on the resulting reversible or quantum circuits. To this end, we implemented the proposed synthesis approach in C++ on top of the BDD package CUDD [19] and synthesized circuits with the respective techniques enabled or disabled.

As benchmarks we used functions provided by RevLib [22] (including most of the functions which have been previously used to evaluate existing reversible synthesis approaches) as well as from the LGSynth package (a benchmark suite for evaluating irreversible synthesis). All experiments have been carried out on an AMD Athlon 3500+ with 1 GB of memory.

5.1 Shared Nodes

To investigate the effect of shared nodes, we extended CUDD so that the application of shared nodes can be disabled or enabled (this has been done by manipulating the unique table). Then, the substitutions of Table 1 and Figure 3 are applied as appropriate.

The results are summarized in Table 4. The first two columns give the name of the benchmark (NAME) as well as the number of primary inputs and outputs (PI/PO). Then, the number of circuit lines (LINES), Toffoli gates (TOF. GATES) or elementary quantum gates (QUA. GATES), as well as the run-time of the synthesis approach (in CPU seconds) is given for the naive approach (w/o SHARED NODES) and the approach that exploits shared nodes (WITH SHARED NODES).

One can clearly conclude, that the application of shared nodes leads to better realizations for reversible and quantum logic. Both, the number of lines and the number of gates can be significantly reduced. In particular, for the number of lines this might be unobvious since additional lines are required to support shared nodes (see Section 4.1). But due to the fact, that shared nodes also decrease the number of terminal nodes (which require additional lines as well), this effect is compensated.

Table 4
Effect of shared nodes

FUNCTION		W/O SHARED NODES				WITH SHARED NODES			
NAME	PI/PO	LINES	TOF. GATES	QUA. GATES	TIME	LINES	TOF. GATES	QUA. GATES	TIME
REVLIB FUNCTIONS									
3_17_6	3/3	12	22	52	<0.01	10	20	50	<0.01
4_49_7	4/4	21	51	123	<0.01	18	45	114	<0.01
4mod5_8	4/1	9	13	36	<0.01	9	13	36	<0.01
aj-e11_81	4/4	20	46	114	<0.01	19	45	113	<0.01
alu_9	5/1	15	30	73	<0.01	14	29	72	<0.01
decod24-enable_32	3/4	10	10	35	<0.01	9	9	30	<0.01
decod24_10	2/4	7	7	21	<0.01	7	7	21	<0.01
ex-1_82	3/3	8	13	31	<0.01	8	13	31	<0.01
fredkin_3	3/3	5	6	16	<0.01	5	6	16	<0.01
graycode6_11	6/6	49	135	327	<0.01	16	20	45	<0.01
ham3_28	3/3	11	19	47	0.01	10	18	46	<0.01
ham7_29	7/7	75	231	595	<0.01	36	88	224	<0.01
hwb5_13	5/5	36	105	277	<0.01	32	91	238	<0.01
hwb6_14	6/6	68	239	618	<0.01	53	167	437	<0.01
hwb7_15	7/7	136	526	1353	<0.01	84	284	744	<0.01
hwb8_64	8/8	277	1132	2903	0.02	129	456	1195	<0.01
millier_5	3/3	9	18	43	<0.01	8	15	38	<0.01
mini-alu_84	4/2	12	21	57	<0.01	11	20	52	<0.01
mod5d2_17	5/5	31	78	188	<0.01	19	42	102	<0.01
one-two-three_27	3/3	10	16	47	<0.01	10	14	40	<0.01
peres_4	3/3	8	14	37	<0.01	7	9	24	<0.01
rd32_19	3/2	9	16	38	<0.01	8	15	37	<0.01
rd53_68	5/3	31	85	212	<0.01	20	49	130	<0.01
rd73_69	7/3	86	301	730	<0.01	38	105	272	<0.01
rd84_70	8/4	194	679	1650	0.01	52	140	373	<0.01
sym6_63	6/1	23	57	126	0.01	17	34	83	<0.01
sym9_71	9/1	104	325	724	<0.01	35	79	201	<0.01
LGSYNTH FUNCTIONS									
9sym	9/1	104	325	724	<0.01	35	79	201	<0.01
bw	5/28	125	381	935	0.01	97	286	747	<0.01
rd84	8/4	117	419	1053	0.01	50	138	367	<0.01
sqrt8	8/4	44	114	242	<0.01	32	84	188	<0.01
table3	14/14	841	2586	7140	0.03	689	2143	5924	0.02
xor5	5/1	17	40	98	<0.01	10	19	48	<0.01

5.2 Complement Edges

Complement edges are supported by the CUDD package and can be easily disabled and enabled. For comparison, we synthesized circuits from both, BDDs with and BDDs without complement edges (denoted by WITH COMPL. EDGES and w/o COMPL. EDGES, respectively). In the latter case, the substitutions shown in Table 3 are applied whenever a successor is connected by a complement edge. Both BDD

Table 5
Effect of Complement Edges

FUNCTION		w/o COMPL. EDGES				WITH COMPL. EDGES			
NAME	PI/PO	LINES	TOF. GATES	QUA. GATES	TIME	LINES	TOF. GATES	QUA. GATES	TIME
REVLIB FUNCTIONS									
3.17_6	3/3	10	20	50	<0.01	8	17	33	<0.01
4.49_7	4/4	18	45	114	<0.01	16	45	97	<0.01
4mod5_8	4/1	9	13	36	<0.01	8	16	37	<0.01
aj-e11_81	4/4	19	45	113	<0.01	16	43	105	<0.01
alu_9	5/1	14	29	72	<0.01	11	25	53	<0.01
decod24-enable_32	3/4	9	9	30	<0.01	9	14	33	<0.01
decod24_10	2/4	7	7	21	<0.01	6	11	23	<0.01
ex-1_82	3/3	8	13	31	<0.01	7	14	32	<0.01
fredkin_3	3/3	5	6	16	<0.01	5	6	16	<0.01
graycode6_11	6/6	16	20	45	<0.01	11	15	15	<0.01
ham3_28	3/3	10	18	46	<0.01	6	12	22	<0.01
ham7_29	7/7	36	88	224	<0.01	18	50	82	<0.01
hwb5_13	5/5	32	91	238	<0.01	27	85	201	<0.01
hwb6_14	6/6	53	167	437	<0.01	46	157	377	<0.01
hwb7_15	7/7	84	284	744	<0.01	74	276	665	<0.01
hwb8_64	8/8	129	456	1195	<0.01	116	442	1067	<0.01
miller_5	3/3	8	15	38	<0.01	8	16	39	<0.01
mini-alu_84	4/2	11	20	52	<0.01	10	22	49	<0.01
mod5d2_17	5/5	19	42	102	<0.01	12	28	49	<0.01
one-two-three_27	3/3	10	14	40	<0.01	9	16	35	<0.01
peres_4	3/3	7	9	24	<0.01	5	7	11	<0.01
rd32_19	3/2	8	15	37	<0.01	6	10	19	<0.01
rd53_68	5/3	20	49	130	<0.01	13	34	75	<0.01
rd73_69	7/3	38	105	272	<0.01	25	73	162	<0.01
rd84_70	8/4	52	140	373	<0.01	34	104	229	<0.01
sym6_63	6/1	17	34	83	<0.01	14	29	69	<0.01
sym9_71	9/1	35	79	201	<0.01	27	62	153	<0.01
LGSYNTH FUNCTIONS									
9sym	9/1	35	79	201	<0.01	27	62	153	<0.01
bw	5/28	97	286	747	<0.01	91	317	732	<0.01
clip	9/5	172	597	1544	<0.01	152	584	1397	0.01
cordic	23/2	76	177	448	0.02	53	109	265	0.02
ex5p	8/63	276	680	1676	0.02	233	706	1520	0.02
pdc	16/40	648	2074	4844	0.12	631	2109	4803	0.12
rd84	8/4	50	138	367	<0.01	33	103	226	<0.01
spla	16/46	567	1422	3753	0.09	559	1728	3799	0.09
sqrt8	8/4	32	84	188	<0.01	30	87	183	<0.01
table3	14/14	689	2143	5924	0.02	686	2413	5926	0.01
xor5	5/1	10	19	48	<0.01	6	8	8	<0.01

types apply shared nodes since their application has been shown to be beneficial (see above). The results are given in Table 5⁴. The columns are labeled as described in Section 5.1.

Even if the cascades representing nodes with complement edges are larger in some cases (see Section 4.2), improvements in the circuit sizes can be observed (see e.g. *rd84_70*, *9sym*, or *cordic*). But in particular for the LGSynth functions often better Toffoli circuits result, when complement edges are disabled (see e.g. *seq*, *spla*, or *table3*). Here, the larger cascades obviously cannot be compensated by complement edge optimizations. In contrast, for quantum circuits in nearly all cases better realizations are obtained with complement edges enabled. A reason for that is, that the quantum cascades for nodes with complement edges have the same size as the respective cascades for nodes without complement edges in nearly all cases (see Table 1, Figure 3 and Table 3, respectively). Thus, the advantage of complement edges (namely the possibility to create smaller BDDs) can be fully exploited without the drawback that the respective gate substitutions become larger.

5.3 Ordering of BDDs

To evaluate the effect of the BDD orderings on the resulting circuit sizes, three techniques are considered: (1) An ordering given by the occurrences of the primary inputs in the function to be synthesized (denoted by ORIGINAL), (2) an optimized ordering achieved by sifting [17] (denoted by SIFTING), and (3) an exact ordering [8] which ensures the BDD to be minimal (denoted by EXACT). Again, all created BDDs exploit shared nodes. Furthermore, complement edges are enabled in this evaluation. After applying our synthesis approach, circuit sizes as summarized in Table 6 result. Here again, the columns are labeled as described in Section 5.1.

The results show, that the ordering has a strong effect on the circuit size. In particular for the LGSynth functions, the best results are achieved with the exact ordering. But as a drawback, this requires a longer run-time. Besides that, also in this evaluation, examples can be found, showing that optimization for BDDs not always leads to smaller circuits (see e.g. *ham7_29* or *hwb7_15* where the best results are achieved with the naive ordering). But in most of the cases improvements are observed. In comparison to previous work, for the first time functions with more than 30 variables can be synthesized.

6 Conclusions and Future Work

In this paper, we described and evaluated how optimization techniques for decision diagrams can also be exploited for BDD-based synthesis of reversible and quantum logic. We considered shared nodes, complement edges, as well as ordering strategies and present the gate cascades needed to support these methods. In a case study the effect of these techniques on the circuit sizes has been evaluated. In most of the cases, BDD optimizations lead to improvements in the circuit sizes as well.

⁴ Compared to Table 4, also benchmarks are considered for which no result could be determined using the w/o SHARED NODES approach.

Table 6
Effect of Variable Ordering

FUNCTION		ORIGINAL				SIFTING				EXACT			
NAME	PI/PO	LINES	TOF. GATES	QUA. GATES	TIME	LINES	TOF. GATES	QUA. GATES	TIME	LINES	TOF. GATES	QUA. GATES	TIME
REVLIB FUNCTIONS													
3.17_6	3/3	8	17	33	<0.01	7	17	29	<0.01	7	17	29	<0.01
4.49_7	4/4	16	45	97	<0.01	15	42	92	<0.01	15	42	92	<0.01
4mod5_8	4/1	8	16	37	<0.01	7	8	18	<0.01	7	8	18	<0.01
aj-e11_81	4/4	16	43	105	<0.01	16	42	96	0.01	15	38	84	<0.01
alu_9	5/1	11	25	53	<0.01	7	9	22	<0.01	7	9	22	<0.01
decod24-enable_32	3/4	9	14	33	<0.01	9	14	33	<0.01	9	14	33	<0.01
decod24_10	2/4	6	11	23	<0.01	6	11	23	<0.01	6	11	23	<0.01
ex-1_82	3/3	7	14	32	<0.01	5	7	17	<0.01	5	7	17	<0.01
fredkin_3	3/3	5	6	16	<0.01	5	6	16	<0.01	5	6	16	<0.01
graycode6_11	6/6	11	15	15	<0.01	11	15	15	0.01	11	15	15	<0.01
ham3_28	3/3	6	12	22	<0.01	7	14	27	0.01	7	14	27	<0.01
ham7_29	7/7	18	50	82	<0.01	21	61	107	<0.01	21	61	107	0.01
hwb5_13	5/5	27	85	201	<0.01	28	88	205	0.01	28	88	205	0.01
hwb6_14	6/6	46	157	377	<0.01	46	159	375	<0.01	46	159	375	0.01
hwb7_15	7/7	74	276	665	<0.01	73	281	653	<0.01	76	278	658	0.01
hwb8_64	8/8	116	442	1067	<0.01	112	449	1047	<0.01	114	440	1051	0.03
millier_5	3/3	8	16	39	<0.01	8	16	39	<0.01	8	16	39	<0.01
mini-alu_84	4/2	10	22	49	<0.01	10	20	43	<0.01	10	20	43	<0.01
mod5d2_17	5/5	12	28	49	<0.01	11	20	30	<0.01	11	20	30	<0.01
one-two-three_27	3/3	9	16	35	<0.01	9	16	35	<0.01	9	16	35	<0.01
peres_4	3/3	5	7	11	<0.01	5	7	11	<0.01	5	7	11	<0.01
rd32_19	3/2	6	10	19	<0.01	6	10	19	0.01	6	10	19	<0.01
rd53_68	5/3	13	34	75	<0.01	13	34	75	<0.01	13	34	75	<0.01
rd73_69	7/3	25	73	162	<0.01	25	73	162	<0.01	25	73	162	<0.01
rd84_70	8/4	34	104	229	<0.01	34	104	229	<0.01	34	104	229	<0.01
sym6_63	6/1	14	29	69	<0.01	14	29	69	<0.01	14	29	69	<0.01
sym9_71	9/1	27	62	153	<0.01	27	62	153	<0.01	27	62	153	<0.01
LGSYNTH FUNCTIONS													
9sym	9/1	27	62	153	<0.01	27	62	153	<0.01	27	62	153	0.01
bw	5/28	91	317	732	<0.01	87	307	693	<0.01	84	306	667	<0.01
clip	9/5	152	584	1397	0.01	66	228	508	<0.01	57	185	392	0.04
cordic	23/2	53	109	265	0.02	52	101	247	0.03	50	95	237	6.90
ex5p	8/63	233	706	1520	0.02	206	647	1388	0.02	206	647	1388	0.06
pdc	16/40	631	2109	4803	0.12	619	2080	4781	0.13	619	2087	4850	66.38
rd84	8/4	33	103	226	<0.01	33	103	226	0.01	33	103	226	<0.01
spla	16/46	559	1728	3799	0.09	489	1709	4372	0.09	483	1687	4322	86.92
sqrt8	8/4	30	87	183	<0.01	30	76	179	<0.01	27	71	173	<0.01
table3	14/14	686	2413	5926	0.01	554	1988	4679	0.02	529	1873	4507	9.88
xor5	5/1	6	8	8	<0.01	6	8	8	<0.01	6	8	8	<0.01

In future work, we plan to adjust the optimization techniques for the synthesis purpose with respect to the expected circuit size, not to the BDD size. As an example, the cost function which is used during reordering should be modified for this purpose. Besides that, also other decompositions should be considered.

References

- [1] Bennett, C. H., *Logical reversibility of computation*, IBM J. Res. Dev **17** (1973), pp. 525–532.
- [2] Brace, K., R. Rudell and R. Bryant, *Efficient implementation of a BDD package*, in: *Design Automation Conf.*, 1990, pp. 40–45.
- [3] Bryant, R., *Graph-based algorithms for Boolean function manipulation*, IEEE Trans. on Comp. **35** (1986), pp. 677–691.
- [4] Cuykendall, R. and D. R. Andersen, *Reversible optical computing circuits*, Optics Letters **12** (1987), pp. 542–544.
- [5] Fredkin, E. F. and T. Toffoli, *Conservative logic*, International Journal of Theoretical Physics **21** (1982), pp. 219–253.
- [6] Gupta, P., A. Agrawal and N. Jha, *An algorithm for synthesis of reversible logic circuits*, IEEE Trans. on CAD **25** (2006), pp. 2317–2330.
- [7] Hung, W., X. Song, G. Yang, J. Yang and M. Perkowski, *Optimal synthesis of multiple output Boolean functions using a set of quantum gates by symbolic reachability analysis.*, IEEE Trans. on CAD **25** (2006), pp. 1652–1663.
- [8] Jeong, S.-W., T.-S. Kim and F. Somenzi, *An efficient method for optimal BDD ordering computation*, in: *International Conference on VLSI and CAD*, 1993, pp. 252–256.
- [9] Kerntopf, P., *A new heuristic algorithm for reversible logic synthesis*, in: *Design Automation Conf.*, 2004, pp. 834–837.
- [10] Landauer, R., *Irreversibility and heat generation in the computing process*, IBM J. Res. Dev. **5** (1961), p. 183.
- [11] Maslov, D., G. W. Dueck and D. M. Miller, *Toffoli network synthesis with templates*, IEEE Trans. on CAD **24** (2005), pp. 807–817.
- [12] Maslov, D., G. W. Dueck and D. M. Miller, *Techniques for the synthesis of reversible toffoli networks*, ACM Trans. on Design Automation of Electronic Systems **12** (2007).
- [13] Merkle, R. C., *Reversible electronic logic using switches*, Nanotechnology **4** (1993), pp. 21–40.
- [14] Miller, D. M., D. Maslov and G. W. Dueck, *A transformation based algorithm for reversible logic synthesis*, in: *Design Automation Conf.*, 2003, pp. 318–323.
- [15] Nielsen, M. and I. Chuang, “Quantum Computation and Quantum Information,” Cambridge Univ. Press, 2000.
- [16] Peres, A., *Reversible logic and quantum computers*, Phys. Rev. A (1985), pp. 3266–3276.
- [17] Rudell, R., *Dynamic variable ordering for ordered binary decision diagrams*, in: *Int’l Conf. on CAD*, 1993, pp. 42–47.
- [18] Shende, V. V., A. K. Prasad, I. L. Markov and J. P. Hayes, *Synthesis of reversible logic circuits*, IEEE Trans. on CAD **22** (2003), pp. 710–722.
- [19] Somenzi, F., “CUDD: CU Decision Diagram Package Release 2.3.1,” University of Colorado at Boulder, 2001.
- [20] Toffoli, T., *Reversible computing*, in: W. de Bakker and J. van Leeuwen, editors, *Automata, Languages and Programming*, Springer, 1980 p. 632.
- [21] Wille, R. and R. Drechsler, *BDD-based synthesis of reversible logic for large functions*, in: *Design Automation Conf.*, 2009.
- [22] Wille, R., D. Große, L. Teuber, G. W. Dueck and R. Drechsler, *RevLib: An online resource for reversible functions and reversible circuits*, in: *Int’l Symp. on Multi-Valued Logic*, 2008, pp. 220–225, RevLib is available at <http://www.revlib.org>.
- [23] Wille, R., H. M. Le, G. W. Dueck and D. Große, *Quantified synthesis of reversible logic*, in: *Design, Automation and Test in Europe*, 2008, pp. 1015–1020.