Synthesis of Reversible Logic

Abstract— **A function is reversible if each input vector produces a unique output vector. Reversible functions find many applications, especially in low power design, quantum computing, optical computing, and nanotechnology. Logic synthesis for reversible circuits differs substantially from traditional logic synthesis and is an active field of research at the moment. In this paper, we present the first practical synthesis algorithm and tool for reversible functions with a large number of inputs. It uses positive-polarity Reed-Muller decomposition at each stage to synthesize the function as a network of Toffoli gates. The heuristic uses a priority queue based search tree. It explores candidate factors at each stage in order of their attractiveness, leading to a fast synthesis algorithm. The algorithm produces near-optimal results for the examples discussed in the literature. The key contribution of the work is that the heuristic finds very good solutions for reversible functions with a large number of variables.**

I. Introduction

Today, power reduction has become the main concern for digital logic designers after performance. Landauer [1] proved that power loss is an integral feature of irreversible circuits that have information loss irrespective of the technology the circuit is implemented in. Also, Bennett [2] showed that in order to keep a circuit from dissipating any power, it had to be composed of reversible gates. A reversible gate has a one-to-one mapping between the input and output vectors. For example, the two-input *OR* gate is not reversible because it maps both 01 and 11 to the same output of 1. In contrast, the two-input two-output Toffoli gate (p = a, q = a $\oplus$ b) is reversible because each output vector corresponds to a unique input vector. Quantum gates are reversible [3] by nature, which provides a powerful motivation to study circuits composed of reversible gates.

Reversible gates have been designed in several different technologies such as CMOS [4], nanotechnology [5], and optical [6]. However, there are three major differences between reversible logic synthesis and the logic synthesis that we are accustomed to. First of all, reversible circuits must have an equal number of inputs and outputs. Secondly, reversible logic does not allow fanout. Lastly, reversible circuits are constrained to be acyclic. These differences prevent us from applying traditional methods of logic design. An optimal algorithm, which uses exhaustive search [7] exists, but is too slow to be used in practice. Several different heuristics have been presented in [8]–[11]. Unfortunately, these heuristics do not scale well for large numbers of inputs. They often get stuck in local minima and require extensive use of template matching to improve the quality of the solution.

In this paper, we present Reed-Muller Reversible Logic Synthesizer (RMRLS), a tool for reversible logic synthesis which uses an XOR sum of products decomposition of the output function to synthesize the circuit. Use of the Reed-Muller expansion of the function was also suggested in [12]. However, their algorithm simply uses as many gates as there are terms in the Reed-Muller expansion of the function with the result produced on an additional output line. Such a method fails to take advantage of shared functionality among multi-output functions. Our algorithm, however, is unique because instead of using the original input variables and their complements at each stage, it decomposes the output function in terms of the inputs at the current stage. The key characteristics of our algorithm are as follows.

- It minimizes the number of gates as the primary objective and the size of the gates as the secondary objective.
- It is near-optimal for all 40,320 reversible functions of three variables.
- It is applicable to functions with large numbers of inputs.
- It does not use the variables and their complements at each stage of the circuit, thus reducing circuit size.
- It requires zero extra garbage lines (such lines are required to equalize the number of inputs and outputs).
- It does not require output permutation.

The rest of the paper is organized as follows. Background on Toffoli gates and Reed-Muller expansions is given in Section II. The synthesis algorithm is described in Section III. Section IV contains experimental results obtained when our algorithm is applied to all 8! reversible functions of three variables and other examples from the literature. It also targets several reversible functions with a large number of inputs. Finally, Section V concludes the paper.

c	b	a	c_o	b_o	a_o
0	0	0	0	0	1
0	0	1	0	0	0
0	1	0	1	1	1
0	1	1	0	1	0
1	0	0	0	1	1
1	0	1	1	0	0
1	1	0	1	0	1
1	1	1	1	1	0

Fig. 1. A reversible function

II. Background

In this section, we present some of the preliminary concepts.

Reversible functions: A function is reversible if it maps each input vector to a unique output vector. It is obvious that the number of inputs must be equal to the number of outputs for

c	b	a	c_o	s_o	p_o
0	0	0	0	0	0
0	0	1	0	1	1 *
0	1	0	0	1	1 *
0	1	1	1	0	0
1	0	0	0	1	0
1	0	1	1	0	1 *
1	1	0	1	0	1 *
1	1	1	1	1	0

Fig. 2. Augmented full-adder

d	c	b	a	c_o	s_o	p_o	g_o
0	0	0	0	0	0	0	0
0	0	0	1	0	1	1	1
0	0	1	0	0	1	1	0
0	0	1	1	1	0	0	1
0	1	0	0	0	1	0	0
0	1	0	1	1	0	1	1
0	1	1	0	1	0	1	0
0	1	1	1	1	1	0	1

Fig. 3. Reversible specification for augmented full-adder

such a mapping to exist. A reversible function of n variables can be defined either as a truth table (see Fig. 1) or as a mapping of integers $\{0,1,...,2^n-1\}$ onto itself. For example, the function in Fig. 1 could also be represented as $\{1, 0, 7, 2, 3, 4, 5, 6\}$.

An *irreversible* function can be converted into a reversible function easily. Extra outputs, called garbage outputs, must be added to make the input-output mapping unique. If the maximum number of identical output vectors is p, then the total number of garbage outputs needed is equal to $\lceil log_2 p \rceil$. Of course, constant inputs must then be added as necessary to balance the number of inputs and outputs. Consider the augmented full-adder which produces carry (c_o), sum (s_o), and propagate (p_o) signals. The truth table for this function is shown in Fig. 2. The function is not reversible because there are repeated output vectors (marked with * in the figure). An extra garbage output (g_o) set equal to any of the input bits will make the mapping unique. A constant input d must also be added to make the number of input and output bits equal. Fig. 3 displays the reversible specification for the function. This specification is identical to the one in [13]. Optimal assignment of garbage outputs in order to minimize the resultant circuit is a hard problem which has not been solved yet.

Reversible gates: A reversible gate is a gate that produces a reversible function. There are two main types of reversible gates, the Toffoli gate [14] and the Fredkin gate [15], as defined next.

Toffoli gate: An $n \times n$ Toffoli gate, denoted by $TOFn(x_1, x_2, ..., x_n)$, passes the first $n-1$ inputs (referred to as *control* bits) to the output unchanged and inverts the n^{th} input (referred to as the *target* bit) if the first $n-1$ inputs are all 1. Using x for input and y for output:

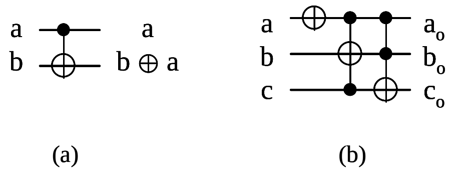

Fig. 4. (a) 2×2 Toffoli gate, and (b) circuit for the function in Fig. 1

$$y_i = x_i \text{ for } 1 \leq i < n$$

$$y_n = x_n \oplus x_1 x_2 .....x_{n-1}$$

A 1×1 Toffoli gate simply inverts the input unconditionally. Fig. 4 presents a graphical representation of a 2×2 Toffoli gate and the implementation of the function in Fig. 1 using Toffoli gates.

Fredkin gate: A three-input Fredkin gate passes the first line to the output and swaps the next two lines if the first input is 0. We will not be using the Fredkin gate in our synthesis.

Reed-Muller expansion: Any Boolean function can be described as an XOR sum of products [16], [17].

The *positive-polarity Reed-Muller (PPRM)* expansion uses only uncomplemented variables and can be derived easily from the function's sum of products expansion. The PPRM of a function is unique and of the form:
$f(x_1, x_2, ..., x_n) = a_0 \oplus a_1 x_1 \oplus a_2 x_2 \oplus ... \oplus a_n x_n \oplus a_{12} x_1 x_2 \oplus a_{13} x_1 x_3 \oplus ... \oplus a_{n-1,n} x_{n-1} x_n \oplus ... \oplus a_{12...n} x_1 x_2 ... x_n$, where $a_i \in \{0, 1\}$ and x_i are all uncomplemented (positive polarity).

The PPRM expansion of the function in Fig. 1 is:
$a_o = a \oplus 1$
$b_o = b \oplus c \oplus ac$
$c_o = b \oplus ab \oplus ac$

The *fixed-polarity Reed-Muller (FPRM)* expansion uses either the uncomplemented variable or the complemented variable, but not both. There are 2^n different fixed-polarity expansions of an n-input function. Determining which one of the 2^n FPRM representations of a function is minimal remains a challenging problem. The FPRM of a function is of the form:
$f(x_1, x_2, ..., x_n) = a_0 \oplus a_1 \dot{x}_1 \oplus a_2 \dot{x}_2 \oplus ... \oplus a_n \dot{x}_n \oplus a_{12} \dot{x}_1 \dot{x}_2 \oplus a_{13} \dot{x}_1 \dot{x}_3 \oplus ... \oplus a_{n-1,n} \dot{x}_{n-1} \dot{x}_n \oplus ... \oplus a_{12...n} \dot{x}_1 \dot{x}_2 ... \dot{x}_n$, where $a_i \in \{0, 1\}$ and $\dot{x}_i$ are all either uncomplemented or complemented.

III. The Synthesis Algorithm

In this section, we describe our synthesis algorithm.

```
Reversible logic synthesis algorithm
Input: function $f$ which is to be synthesized

bestfound ← ∞
curnode ← PPRM expansion of function
init_count ← number of terms in the PPRM expansion of f
Add curnode to priority queue (PQ)
while (curnode = PQ.pop() != NULL) {
    if (curnode.depth() ≥ bestfound−1)
        continue
    foreach (v in variable) {
        foreach (factor in v.expansion()) {
            if (factor contains v)
                continue
            newnode ← curnode
            Substitute v = v ⊕ factor in all expansions in newnode
            if (synthesized(newnode)) {
                bestfound ← newnode.depth()
                cache solution
                break
            }
            else if (newnode.depth() ≥ bestfound−1)
                continue
            else if (evaluate(newnode)) {
                calculate newnode_priority
                PQ.push(newnode, newnode_priority)
            }
        }
    }
}
```

Fig. 5. Synthesis algorithm

A. Basic Algorithm

Consider a reversible function *f* with *n* inputs and *n* outputs. Fig. 5 gives the pseudo-code for the main steps in the algorithm whose primary objective is to minimize the number of Toffoli gates in the circuit, and secondary objective is to minimize the size of the gates. The first stage consists of initialization and setup. The variable *bestfound*, which will store the depth of the best circuit synthesized for *f*, is set to infinity. Next, a Reed-Muller expansion of all the output variables $v_{out,i}$ is obtained in terms of all the input variables v_j and stored in *curnode*. The Reed-Muller expansion for a function can be obtained using a Kronecker matrix [16]. The total number of terms in the Reed-Muller expansion of *f* is stored in *init_count*. Lastly, *curnode* is pushed onto the priority queue.

In the second phase, the algorithm enters a loop, where it pops a node from the priority queue and stores it in *curnode*. If the depth of *curnode* is greater than or equal to $bestfound - 1$, then we disregard *curnode* because it cannot possibly lead to a better solution than the one that has already been found. For each output variable $v_{out,i}$ in *f* that contains the input variable v_i, we search for factors in the Reed-Muller expansion of $v_{out,i}$ contained in *curnode* that do not contain v_i. For example, if $a_{out} = a \oplus 1 \oplus bc \oplus ac$, then the appropriate factors of a_{out} are 1 and bc, as neither contains literal a. For each factor *fac* and output variable $v_{out,i}$ identified in this manner, we make a copy of *curnode* in *newnode*. We next substitute $v_i = v_i \oplus fac$ in the expansions of all the variables contained in *newnode*. We then examine *newnode* and pursue one of the following:

- If the synthesis of the function has been completed (the expansion for all $v_{out,i}$ only contains v_i), we update the value of *bestfound* and cache the solution found.
- Else if the depth of *newnode* is greater than or equal to $bestfound - 1$, we disregard *newnode* (as before, it cannot possibly lead to a better solution than the current one) and move on to the next factor.
- Else we decide whether *newnode* presents an attractive path to follow for synthesis. The function *evaluate* uses several parameters such as the number of inputs of *f*, the current depth, and whether a solution has been found at that point to decide whether to continue the search along the current path. For example, at the top of the search tree, the algorithm explores several paths even if they are not very attractive to make sure different parts of the search space are explored. Depending on the decision, we insert *newnode* into the priority queue with a priority of:
  $newnode\_priority = \alpha * newnode.depth() + \beta * (init\_count - newnode.term\_count) / newnode.depth() - \gamma * (number\ of\ literals(fac))$
  The first term gives preference to nodes of larger depth as all things being equal they are more likely to be close to the solution. The second term addresses the

primary objective of minimizing the number of gates. The average number of terms eliminated per stage is used to measure a node's effectiveness. The third term addresses the secondary objective of minimizing the size of individual gates. The weights α, β, and γ add up to 1. Typical values of α, β, and γ were 0.2, 0.7, and 0.1 respectively.

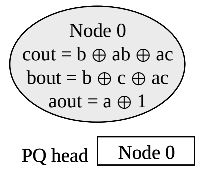

Fig. 6. Applying algorithm to function in Fig. 1: Step 1

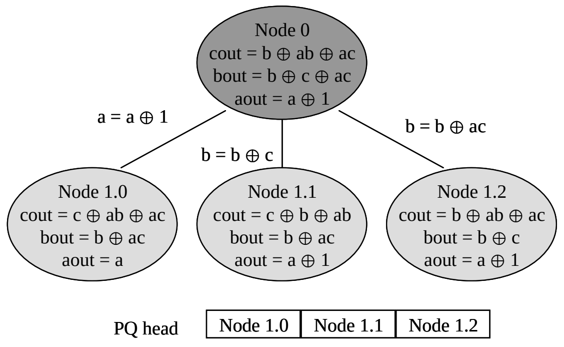

Fig. 7. Applying algorithm to function in Fig. 1: Step 2

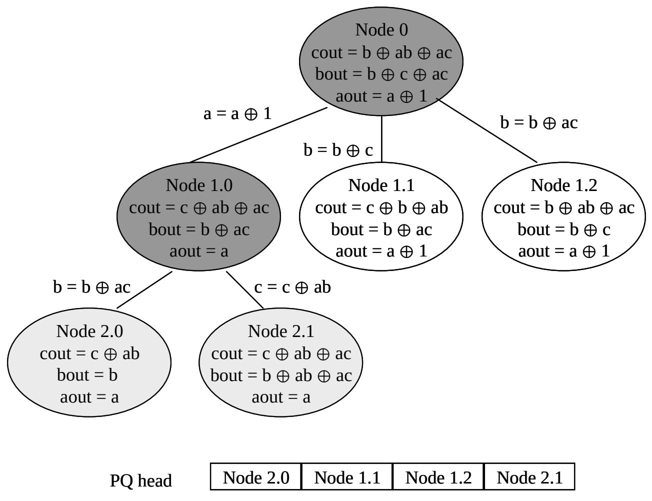

Fig. 8. Applying algorithm to function in Fig. 1: Step 3

Figs. 6 - 8 illustrate the application of the algorithm to the function in Fig. 1. In the figures, the dark shaded nodes have already been explored, lightly shaded nodes have been added in the current stage, and nodes with no shading are yet to be considered. In the first step, the PPRM expansion of the function is stored in *Node0* which is inserted into the priority queue. Fig. 6 shows the search tree and priority queue at this point. In the next step, *Node0* is popped from the priority queue (it is the only item present) and examined for possible substitutions. The algorithm identifies three possible substitutions: $a = a \oplus 1, b = b \oplus c$, and $b = b \oplus ac$. For each substitution we create a new node, substitute the factor identified in all the expansions, and add the nodes to the priority queue. Due to fewer terms in the expansion, *Node1.0* has a higher priority than *Node1.1* and *Node1.2* (Fig. 7). In the next step, *Node1.0* is popped from the priority queue. $b = b \oplus ac$ and $c = c \oplus ab$ are the two possible substitutions from this node. *Node2.0* and *Node2.1* corresponding to these substitutes are inserted into the priority queue with *Node2.0* at the head of the queue with the highest priority (see Fig. 8). In the next iteration through the loop, *Node2.0* is popped from the priority queue. The only possible transformation is $c = c \oplus ab$ which leads to a solution. The solution and its depth are cached. After this point, no new nodes are inserted into the priority queue. *Node1.1* and *Node1.2* are popped in that order respectively, but fail to give solutions from any of their substitutions. Their children are not added to the queue because we have already found a solution of depth three which they will not be able to beat. Lastly, *Node2.1* is popped but discarded because its depth is too large to be useful. The cached solution (shown in Fig. 4(b)) is the best solution that the algorithm finds.

B. *Advanced Features*

In theory, the PPRM expansion can be very large in the worst case. In order to prevent that scenario, we allow two additional types of substitutions.

- We no longer require that the output variable $v_{out,i}$ contain the variable v_i for the algorithm to consider substitutions from that list. For example, in the synthesis for the reversible function in Fig. 1 just presented, the advanced algorithm would also select $c = c \oplus b$ and $c = c \oplus ab$ as possible substitutions in the first stage.
- For any variable v_i, we also allow the substitution $v_i = v_i \oplus 1$ even if the expansion of $v_{out,i}$ does not contain 1. Thus, in the synthesis for the reversible function in Fig. 1, the substitutions $b = b \oplus 1$ and $c = c \oplus 1$ would be added as possible substitutions in the first stage.

IV. Experimental Results

In this section, we present various experimental results. Table I depicts the results of applying various algorithms to all reversible functions of three variables, which number $8!$ or $40,320$. Column *#gates* represents the number of Toffoli gates required for the implementation. Column *Ours* presents results obtained by our algorithm. Synthesis only takes a few minutes on a 1.2GHz Athlon processor with 512 MB RAM. Column *Miller* shows the results obtained by the heuristic in [13], while column *Optimal* contains optimal results from [7]. The last row displays the average gate count for each algorithm. As can be seen, our algorithm produces a lower average gate count

TABLE I
ALL REVERSIBLE FUNCTIONS OF THREE VARIABLES

#gates	Ours	Miller [13]	Optimal [7]
11		5	
10		110	
9	30	729	
8	3297	4726	577
7	12488	11199	10253
6	13620	12076	17049
5	7503	7518	8921
4	2642	2981	2780
3	625	767	625
2	102	130	102
1	12	15	12
0	1	1	1
Ave. size	6.10	6.18	5.87

than Miller's heuristic [13] and does not produce any circuits containing 10 or 11 Toffoli gates. This is inspite of the fact that a larger library of gates is used in [13]. Specifically, they use the SWAP gate which unconditionally exchanges its two inputs when passing them to the outputs. Furthermore, our results compare very favorably to the optimal implementations found using depth-first search with iterative deepening (this is an exhaustive approach).

We now provide results for several examples from the literature. Unless otherwise stated, our results are equal in the number of gates to the best published solution and were synthesized in less than 0.1 seconds.

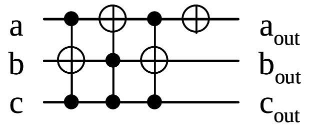

Fig. 9. Example 1 realization

The next two examples are from [13].
Example 1:
Specification: {1, 0, 3, 2, 5, 7, 4, 6}.
Toffoli network produced: TOF3(c,a,b) TOF3(c,b,a) TOF3(c,a,b) TOF1(a).
This means that there are four Toffoli gates cascaded together, as shown in Fig. 9.

Example 2:
Specification: {7, 0, 1, 2, 3, 4, 5, 6}.
Toffoli network produced: TOF1(a) TOF2(a,b) TOF3(b,a,c).
The heuristic in [13] initially synthesized a circuit with seven gates for this specification. This circuit was then improved to a circuit with three gates through bi-directional synthesis (i.e., simultaneous synthesis from inputs towards outputs and outputs towards inputs). Our algorithm produces the same circuit with three gates.

The next series of examples are from [9].
Example 3: This example deals with the realization of a Fredkin gate using Toffoli gates.
Specification: {0, 1, 2, 3, 4, 6, 5, 7}.
Toffoli network produced: TOF3(c,a,b) TOF3(c,b,a) TOF3(c,a,b).

Example 4: This transformation involves a simple swap between two positions in the truth table.
Specification: {0, 1, 2, 4, 3, 5, 6, 7}.
Toffoli network produced: TOF2(c,b) TOF3(c,b,a) TOF3(b,a,c) TOF3(c,b,a) TOF2(c,b).

Example 5: An extension of the last example to four variables.
Specification: {0, 1, 2, 3, 4, 5, 6, 8, 7, 9, 10, 11, 12, 13, 14, 15}.
Toffoli network produced: TOF2(d,b) TOF3(d,b,a) TOF4(d,b,a,c) TOF4(c,b,a,d) TOF4(d,b,a,c) TOF3(d,b,a) TOF2(d,b).

Example 6: This represents a wraparound shift of one position for a three-variable function.
Specification: {1, 2, 3, 4, 5, 6, 7, 0}.
Toffoli network produced: TOF3(b,a,c) TOF2(a,b) TOF1(a).

Example 7: This represents a wraparound shift of one position for four variables.
Specification: {1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 0}.
Toffoli network produced: TOF4(c,b,a,d) TOF3(b,a,c) TOF2(a,b) TOF1(a).

The above examples show that our algorithm matches the best results presented in the literature for specifications with a small number of variables. Due to the inability of previous algorithms to deal with specifications with a large number of variables, such examples do not exist in the literature. However, the main advantage of our algorithm is that it can easily tackle larger functions. The key to successful synthesis of larger functions is the ability to backtrack quickly once a path through the search tree has become inefficient. Our priority-based traversal of the search tree is well-suited to such backtracking. We next consider larger benchmarks.

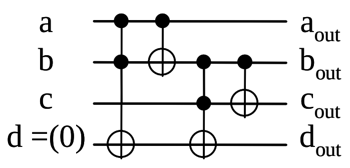

Fig. 10. Full-adder realization

Adder: Consider the reversible full-adder which generates

sum, carry, and propagate signals (discussed in Section I). There is a garbage output set to one of the input bits to make the function reversible, and the constant d input is added to make the number of input and output bits equal.
Specification:

$$d_{out} = d \oplus ab \oplus bc \oplus ac \text{ (carry bit)}$$

$$c_{out} = a \oplus b \oplus c \text{ (sum bit)}$$

$$b_{out} = a \oplus b \text{ (propagate bit)}$$

$$a_{out} = a \text{ (garbage bit)}$$

Toffoli network produced: TOF3(b,a,d) TOF2(a,b) TOF3(c,b,d) TOF2(b,c) (See Fig. 10).

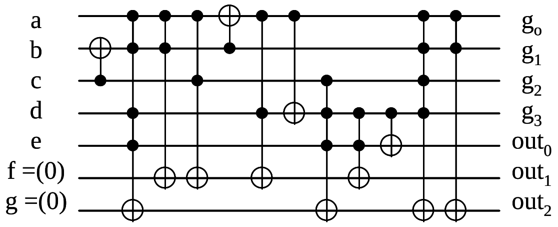

Fig. 11. rd53 realization

rd53: Benchmark rd53 is from the MCNC [18] benchmark suite and has five inputs and three outputs. The output vector is equal to the number of 1s in the input vector. Thus, $\{00000\}$ yields $\{000\}$, $\{00101\}$ yields $\{010\}$, and $\{11111\}$ yields $\{101\}$.
Specification: Output vectors $\{010\}$ and $\{011\}$ both occur ten times and hence we need to add $\lceil log_2 10 \rceil = 4$ garbage outputs for a total of seven outputs. This requires the addition of two more constant inputs to the five existing ones. We used the same reversible specification as in [13].
Toffoli network produced: TOF2(c,b) TOF5(e,d,b,a,g) TOF3(b,a,f) TOF3(c,a,f) TOF2(b,a) TOF3(d,a,f) TOF2(a,d) TOF4(e,d,c,g) TOF3(e,d,f) TOF2(d,e) TOF5(d,c,b,a,g) TOF3(b,a,g) (See Fig. 11).

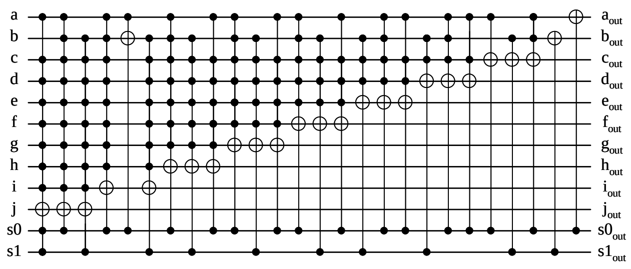

Fig. 12. Shifter realization

Shifter: This function has two control bits and ten input bits. Depending on the value of the two control bits, the function does a wraparound shift of zero, one, two, or three positions on the input. The control bits are passed unchanged to the output. E.g., if control bits are 10, then $\{0,1,2,3,...,2^{10}-1\}$ will become $\{2,3,...,2^{10}-1,0,1\}$. See Fig. 12 for the 26-gate Toffoli network for this function, which was synthesized in 1.2 seconds.

V. Conclusions

We have presented an algorithm which uses a positive-polarity Reed-Muller decomposition of a reversible function to select successive Toffoli gates. The algorithm searches the tree of possible factors in priority order to try to find the best possible solution. The use of extensive pruning leads to very fast synthesis. We applied our algorithm to all 40,320 functions of three variables and obtained near-optimal results. Several examples of functions with a large number of variables were also presented to demonstrate the suitability of the algorithm for synthesizing complex functions.

As part of future work, we would like to incorporate Fredkin gates into our algorithm. A Fredkin gate is equivalent to three Toffoli gates. Thus, the use of Fredkin gates could yield a significant improvement in circuit quality. We are also working on ways to efficiently synthesize functions with *don't cares*. We currently pre-assign values to *don't care* outputs. It would be better if we could find a way to dynamically assign these values during the search phase.

References

[1] R. Landauer, "Irreversibility and heat generation in the computing process," *IBM J. Res.*, vol. 5, pp. 183–191, 1961.
[2] C. Bennett, "Logical reversibility of computation," *IBM J. Res. Dev.*, vol. 17, pp. 525–532, 1973.
[3] M. Nielsen and I. Chuang, *Quantum Computation and Quantum Information*. Cambridge University Press, 2000.
[4] G. Schrom, "Ultra-low-power CMOS Technology," *PhD thesis, Technischen University at Wien*, June 1998.
[5] R. C. Merkle, "Two types of mechanical reversible logic," *Nanotechnology*, vol. 4, pp. 114–131, 1993.
[6] E. Knill, R. Laflamme, and G. J. Milburn, "A scheme for efficient quantum computation with linear optics," *Nature*, pp. 46–52, Jan. 2001.
[7] V. V. Shende, A. K. Prasad, I. L. Markov, and J. P. Hayes, "Reversible logic circuit synthesis," in *Proc. Int. Conf. Computer-Aided Design*, Nov. 2003, pp. 125–132.
[8] D. Maslov and G. W. Dueck, "Garbage in reversible design of multiple output functions," in *Proc. 6th Int. Symp. Representations & Methodology of Future Computing Technologies*, Mar. 2003, pp. 162–170.
[9] D. M. Miller and G. W. Dueck, "Spectral techniques for reversible logic synthesis," in *Proc. 6th Int. Symp. Representations & Methodology of Future Computing Technologies*, Mar. 2003.
[10] K. Iwama, Y. Kambayashi, and S. Yamashita, "Transformation rules for designing CNOT-based quantum circuits," in *Proc. Design Automation Conf.*, June 2002.
[11] A. Khlopotine, M. Perkowski, and P. Kerntopf, "Reversible logic synthesis by iterative compositions," in *Proc. Int. Wkshp. Logic Synthesis*, June 2002.
[12] A. Mishchenko and M. Perkowski, "Logic synthesis of reversible wave cascades," in *Proc. Int. Wkshp. Logic Synthesis*, June 2002.
[13] D. M. Miller, D. Maslov, and G. W. Dueck, "A transformation based algorithm for reversible logic synthesis," in *Proc. Design Automation Conf.*, June 2003.
[14] T. Toffoli, "Reversible computing," *Tech. memo MIT/LCS/TM-151, MIT Lab for Comp. Sci*, 1980.
[15] E. Fredkin and T. Toffoli, "Conservative logic," *Int. J. of Theoretical Physics*, vol. 21, pp. 219–253, 1982.
[16] T. Sasao, *Logic Synthesis and Optimization*. Netherlands: Kluwer Academic Publishers, 1993.
[17] M. Thorton, "Fixed polarity Reed-Muller forms." [Online]. Available: http://engr.smu.edu/~mitch/class/8391/week12/esop.pdf.
[18] N.C.S.U. Collaborative Benchmarking Laboratory, Dept. of Computer Science, North Carolina State University, "Benchmark archives at CBL." [Online]. Available: http://www.cbl.ncsu.edu/benchmarks/