REVS: A Tool for Space-Optimized Reversible Circuit Synthesis

Alex Parent[1,2], Martin Roetteler[2(✉)], and Krysta M. Svore[2]

[1] Institute for Quantum Computing, University of Waterloo, 200 University Avenue West, Waterloo, ON, Canada
alexparent@gmail.com

[2] Quantum Architectures and Computation Group, Microsoft Research, One Microsoft Way, Redmond, WA 98052, USA
{martinro,ksvore}@microsoft.com

Abstract. Computing classical functions is at the core of many quantum algorithms. Conceptually any classical, irreversible function can be carried out by a Toffoli network in a reversible way. However, the Bennett method to obtain such a network in a "clean" form, i.e., a form that can be used in quantum algorithms, is highly space-inefficient. We present REVS, a tool that allows to trade time against space, leading to circuits that have a significantly smaller memory footprint when compared to the Bennett method. Our method is based on an analysis of the data dependency graph underlying a given classical program. We report the findings from running the tool against several benchmarks circuits to highlight the potential space-time tradeoffs that REVS can realize.

1 Introduction

The ability to compute classical functions is at the core of many interesting quantum algorithms, including Shor's algorithm for factoring, Grover's algorithm for unstructured search, and the HHL algorithm for inverting linear systems of equations. While conceptually any classical, irreversible function can be carried out by a reversible Toffoli network, the standard way to obtain such a network is highly space-inefficient: the so-called Bennett method leads to a number of qubits that is proportional to the circuit size of the given classical, irreversible function [3].

We show that it is possible to trade time against space in reversible circuit synthesis, leading to circuits that have a significantly smaller memory footprint than the ones generated by the Bennett method. To this end, we implemented a tool for space-optimized reversible synthesis. We applied our tool to a suite of challenge problems that include a subset of several classical circuits benchmarks such as the ISCAS and MCNC benchmarks, as well as reversible benchmarks such as the Maslov benchmarks and the RevLib benchmarks. We show that it is typically possible to reduce the total number of required ancillas by a factor of $4X$ at a moderate increase of the total number of gates by less than $3X$.

I. Phillips and H. Rahaman (Eds.): RC 2017, LNCS 10301, pp. 90–101, 2017.
DOI: 10.1007/978-3-319-59936-6_7

Prior work. Several tools were developed for synthesizing reversible circuits, ranging from low-level tools [13,15,24,26,30–32], over various optimizations [24], to high-level programming languages and compilers [9–11,22,23,29,33,36]. See also [25] for a survey. We are interested in methods that optimize space, i.e., methods that synthesize target functions while using as few ancillas as possible.

The implied trade-off is between circuit size, as measured by the total number of Toffoli gates, and circuit width, as measured by the total number of qubits. Methods to obtain such trade-offs have been studied in the literature before, notably in the theoretical computer science community where space-time trade-offs based on trading qubits (lines) for gates [7,27,34,35] and tradeoffs based on Bennett's pebble game have been known for quite some time [4–6,12]. Our work implements a compiler that optimizes for space, trading it for a possibly a slightly larger gate count and possibly for a longer compilation time.

Our contribution. We improve the space-efficiency of Toffoli networks by analyzing the data flow dependencies of the given input program or truth table. This allows to clean some of the required ancilla bits much earlier than possible with the Bennett method. Another key component that allowed us to improve the memory footprint while keeping the circuit size of the resulting networks relatively small, is the combination of known techniques for Boolean Exclusive Sum-Of-Products (ESOP) [17,18] minimization with our dependency-graph based methods for early cleanup.

Specifically, we considered the RevLib benchmarks [1] and the Maslov benchmarks [14]. Our main result is that for some of the benchmarks we can improve the total number of qubits needed. This typically comes at an increase of the overall gate count, however, for some of the benchmarks our method achieves an improvement in terms of number of qubits *and* total number of Toffoli gates.

Generally, the methods described in this paper aim at large circuits, i.e., they are *scalable*: our reversible synthesis method starts from high-level descriptions in a functional programming language.

2 Reversible Circuits

Reversible functions are Boolean functions $f : \{0,1\}^n \to \{0,1\}^n$ that can be inverted on all outputs, i.e., the functions that correspond to permutations of a set of cardinality 2^n. As with classical circuits, reversible functions can be constructed from universal gate sets: for instance, it is known that the Toffoli gate which maps $(x, y, z) \mapsto (x, y, z \oplus xy)$, together with the controlled-NOT gate (CNOT) which maps $(x, y) \mapsto (x, x \oplus y)$ and the NOT gate which maps $x \mapsto x \oplus 1$, is universal for reversible computation. The group generated by all NOT, CNOT, and Toffoli gates on $n \geq 4$ bits is isomorphic to the alternating group A_{2^n} of even permutations which is a group of order $(2^n)!/2$. Hence, any given target function, when considered as a permutation π can be implemented over this gate set at the expense of at most 1 additional qubit since $\mathbf{1} \otimes \pi = \text{diag}(\pi, \pi)$ is even.

Most classical functions $f : \{0,1\}^n \to \{0,1\}^m$ are not invertible. To make a classical function reversible a permutation π on a larger space has to be constructed that implements f on a subset of size 2^n of the inputs. These extra bits are commonly denoted *ancilla* bits and are used as *scratch space*, i.e., temporary bits which store intermediate results of a computation. A very important difference to classical computing is that scratch bits cannot just be overwritten when they are no longer needed: any ancilla that is used as scratch space during a reversible computation must be returned to the initial value—which is commonly assumed to be the value 0—computationally.

Moreover, if this return to a "clean" value is not achieved, the function cannot be safely used inside a quantum computer as its use might lead to unwanted entanglement of the computational registers with the ancilla qubits. This in turn can destroy desired interferences crucial for quantum algorithms [20]. If a Toffoli network computes a target function in a way that leaves garbage bits that are unclean, then one can turn this into a clean network using Bennett's method, however, this leads to a 2X increase in circuit size and additional qubits to store the output.

The number of Toffoli gates used in the implementation of a given permutation is the basic measure for the circuit *size* that we use in this paper. Counting Toffolis only is justified from the theory of fault-tolerant quantum computing [20] since the Toffoli gate (and the T gate) has a substantial cost, whereas the cost of so-called Clifford gates, such as CNOT and NOT, can usually be neglected. Another related metric is the overall depth of the circuit, measured usually in the form of T-gate-depth. Implementations of the Toffoli gate over the Clifford+T gate set are known [20]. The other basic parameter in our design space is circuit *width*, measured as the maximum number of qubits needed during any point, i.e., the maximum number of input qubits, output qubits, and ancilla qubits.

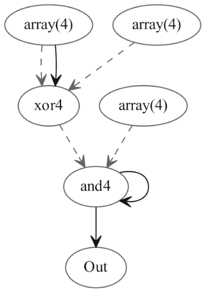
Generally, our goal is to trade time for space, i.e., to achieve a reduction in the total number of qubits required. In turn, we are willing to pay a price in terms of a slight increase in the total number of Toffoli gates and in terms of compilation time. Our trade-off is justified by the limited number of qubits available in experimental quantum devices.

3 Dependency Analysis

Analyzing the dependencies between the instructions in a basic function, between functions, and between larger units of code is a fundamental topic in compiler design [2,19]. Typically, dependency analysis consists of finding codes units and to identify them with nodes in a directed acyclic graph (DAG). The directed edges in the graph are the dependencies between the basic units, i.e., anything that might constrain the execution order, for instance control dependencies that arise from the control flow in the program, or branchings that happen conditional on the value of a variable or the causal dependencies that arise from one unit having to wait for the output of another unit before the computation can proceed.

```
let xor4 (a:bool array) (b:bool array) =
    let c = Array.zeroCreate 4
    for i in 0 .. 3 do
            c.[i] <- a.[i] <> b.[i]
    c
let and4 (a:bool array) (b:bool array) =
    let d = Array.zeroCreate 4
    for i in 0 .. 3 do
        d.[i] <- a.[i] && b.[i]
    d
let mutable a = Array.zeroCreate 4
a <- xor4 a b
and4 a c
```

(a) F# snippet



(b) Corresponding MDD

Fig. 1. (a) Simple F# code example of a function that uses arrays and in place operations. (b) Corresponding mutable data dependency (MDD) graph with data dependency arrows (dashed) and mutation arrows (bold).

3.1 Mutable Data Dependency Graphs (MDDs)

We used the .NET language F# to implement a compiler for a language that can express classical, irreversible functions and turn them into reversible networks. The language itself is also a subset of F# which has the advantage that all programs expressed in the language also have an abstract interpretation as executable programs that can be run on the .NET common language run-time (CLR). This helps with testing of the reversible circuits generated by our compiler as it is possible to (a) generate a Toffoli network from the source program and (b) get a trace from the execution on a classical computer and then to compare (a) and (b).

The compilation itself follows some steps that are common for domain-specific approaches. As our language is embedded into F#, we can first invoke the F# compiler to generate an abstract syntax tree (AST) for the input program. Using so-called active patterns [28] we turn the AST into an internal representation that represents the dependency graph of the program. The nodes of this graph capture the control flow and data dependencies between expressions, but also identify which blocks can be computed by in-place operations and which blocks have to be computed by out-of-place operations. Because of this latter feature is related to which elements of the dependency graph are mutable and which are not, we call this data structure the Mutable Data Dependency graph or MDD.

Which parts of the code can be computed by an in-place operation is inferred by which variables are labeled in F# as `mutable` together with the external knowledge about whether for an expression involving these variables an in-place implementation is actually known. An example for the latter is the addition operation for which we can choose either an in-place implementation $(a, b) \mapsto (a, a + b)$ or an out-of-place implementation $(a, b, 0) \mapsto (a, b, a + b)$.

The nodes of the MDD correspond to inputs, computations, initialized and cleaned-up bits. Inputs nodes can correspond to individual variables but also to entire arrays which are also represented as a single node and treated atomically. Computation nodes correspond to any expression that occurs in the program and that manipulates the data. Initialized and cleaned-up bits correspond to bits that are part of the computation and which can be used either as ancillas or to hold the actual final output of the computation. Initialization implies that those qubits are in the logical state 0 and the cleaned-up state means these bits are known to be returned back in the state 0.

The directed edges in a MDD come in two flavors: data dependencies and mutations. Data dependencies are denoted by dashed arrows and represent any data dependency that one expression might have in relation to any other expression. Mutations are denoted by bold arrows and represent parts of the program that are changed during the computation. By tracking the flow of the mutations one can then ultimately determine the scheduling of the expressions onto reversible operations and re-use a pool of available ancillas. This helps to reduce the space requirements of the computation, in some cases even drastically so.

First, a number of arrays are used to store data in a way that allows for easy access and indexing. Note that in F# the type `array` is inherited from the .NET array type and by definition is a mutable type. This information is used when the MDD for the program is constructed as our compiler knows that in principle the values in the array can be updated and overwritten. Whether this can actually be leveraged when compiling a reversible circuit will of course depend on other factors as well, namely whether the parts of the data that is invoked in assignments (denoted by $<-$) is used at a later stage in the program, in which case the data might have to be recomputed.

When resolving the AST of a function, each node will either be another function or an input variable. If the node is a function, we recursively compute the AST for all of the function inputs adding the results to the graph. Upon doing so, we use the index numbers of these results as the inputs for the operation and then add the operation to the graph. If the node is a variable, the algorithm looks up its name in a map of currently defined variables and returns an index to its node. The type of the operation determines which arrows will be solid input arrows and which will be data dependencies, i.e., controls. An example is shown in Fig. 1.

3.2 Eager Cleanup Strategy

From Bennett's work on reversible Turing machines it follows that any function can be implemented by a suitable reversible circuit [3]: if an n-bit function $x \mapsto f(x)$ can be implemented with K gates over $\{\mathrm{NOT}, \mathrm{AND}\}$, then the reversible function $(x, y) \mapsto (x, y \oplus f(x))$ can be implemented with at most $2K + n$ gates over the Toffoli gate set. The basic idea behind Bennett's method is to replace all AND gates with Toffoli gates, then perform the computation, copy out the result, and undo the computation. One potential disadvantage of Bennett's method is the large number of ancillas it requires as the required memory scales proportional

Algorithm 1. EAGER Performs eager cleanup of an MDD.

```
Require: An MDD $G$ in reverse topological order, subroutines LastDependentNode,
    ModificationPath, InputNodes.
 1: $i \leftarrow 0$
 2: for each node in G do
 3:     if modificationArrows node = $\emptyset$ then
 4:         dIndex ← LastDependentNode of node in $G$
 5:         path ← ModificationPath of node in $G$
 6:         input ← InputNodes of path in G
 7:         if None (modificationArrows input) $\geq$ dIndex then
 8:             cleanUp ← (Reverse path) ++ cleanNode
 9:         end if
10:     else
11:         cleanUp ← uncleanNode
12:         $G$ ← Insert cleanUp Into $G$ After dIndex
13:     end if
14: end for
15: return $G$
```

to the circuit *size* of the initial, irreversible function f. Nevertheless, Bennett's method is useful to clean up garbage qubits in some situations where our improved synthesis method, which we call the "eager cleanup" strategy, does not succeed. The basic idea behind eager cleanup is to process the MDD in inverse topological order and try to clean up qubits that are no longer needed as early as possible. To do this, when we find a node A which does not have an outgoing modification arrow we first find the node furthest along in topological order which depends on it B. We then consider all inputs in the modification path of A. If any of the inputs have outgoing arrows modification arrows pointing levels previous to B we may not clean the bit eagerly as its inputs are no longer available. If the inputs do not have modification arrows pointing at levels previous to B we can immediately clean it up by reversing all operations along its modification path. In many cases, the eager cleanup strategy leads to lower number of qubits used compared to Bennett's original method [3]. A pseudo-code implementation of the eager cleanup strategy is shown in Algorithm 1.

4 Boolean Expression Generation

REVS handles higher-level, irreversible programs using cleanup strategies such as Bennett's method or the eager cleanup strategy mentioned in the previous section. This is particularly useful if the irreversible program has control flow such as loops, branchings, and subroutine calls. If a piece of the given code corresponds to a Boolean expression directly, then synthesis is handled differently: in these cases truth-table based techniques such as the ones described in [15] could be applied, however, in the current implementation we follow a simple flow that takes the Boolean function, either given in BLIF or PLA format, transforms

Algorithm 2. ESOP-FACTOR Find and factor common ESOP expressions.

```
Require: Boolean expression $exprs$ as list of ($input$, $output$) pairs, integer $sizeParam$
    to specify maximum group size.
1: outputGroups ← group $exprs$ with identical output
2: factorGroups ← Divide each group in outputGroups into groups of size $sizeParam$
   or less
3: for each group in factorGroups do
4:     xorExpr ← expression formed by XORing all input expressions in
       group together
5:     factoredExpr ← use multi-level optimization techniques to factor xorExpr
6:     circuit ← apply boolean expression generation algorithm to factoredExpr
7: end for
8: return circuit
```

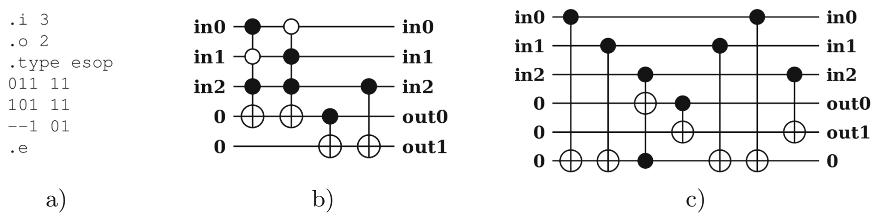
it into exclusive-sum-of-product (ESOP) format using Exorcism [18], and then further process it using strategies that again allow tradeoffs between circuit size and number of qubits used. We briefly sketch these methods next and show the application to some benchmarks used in reversible synthesis.

4.1 Boolean Function Synthesis Benchmarks

The Berkeley Logic Interchange Format (BLIF) and the Programmable Logic Array format (PLA) allow logic level circuit description of a classical operation. Both formats allow the specification of hierarchical logical circuits, based on a simple text input form. Circuits can have combinational components, which typically are given by a collection of truth tables using separate lines for each input/output combinations, where “don’t cares” are allowed. Circuits are also allowed to have sequential components such as latches.

BLIF underlies many circuit benchmarks that have been used primarily by the Circuit and Systems community in the 80s and 90s. These benchmarks include the ISCAS’85, ISCAS’89, MCNC’91, LGSynth’91 and LGSynth’92 collections [16]. We identified all examples from the union of these benchmarks that *only* use combinational circuit elements. For those Boolean functions in principle a reversible circuit can be computed. We obtained a set of 135 benchmark circuits which we used to test the performance of our Boolean generation subroutines. On these circuits we typically found that our tool REVS decreased the number of ancillas by a factor of $4x$ while increasing the number of gates only moderately.

PLA underlies benchmarks for reversible circuit synthesis that typically start as classical, irreversible functions expressed in this format. The two benchmarks we considered are the RevLib benchmarks [1] and the Maslov benchmarks [14]. We optimized the reversible circuits for space using the methods described in this paper and compared it to the best known circuits in the RevLib and Maslov databases. While generally, we get a tradeoff between space and time, in some cases we found circuits that are more efficient in terms of number of qubits *and* the total circuit size.



```
.i 3
.o 2
.type esop
011 11
101 11
--1 01
.e
```

Fig. 2. An example illustrating the synthesis based on factorization of output groups for ESOP files. Shown in (a) is a simple example of an ESOP file corresponding to the output functions $(f_0(x_0, x_1, x_2), f_1(x_0, x_1, x_2))$, where $f_0(x_0, x_1, x_2) = \overline{x_0}x_1x_2 \oplus x_0\overline{x_1}x_2$ and $f_1(x_0, x_1, x_2) = \overline{x_0}x_1x_2 \oplus x_0\overline{x_1}x_2 \oplus x_2$. Running REVS with parameter $p = 0$ treats each line in the ESOP file as a group. This turns each line into a multiply-controlled Toffoli gate as shown in (b). Running REVS with parameter $p = 1$ allows REVS to group up to 2 lines together, provided that the lines have identical outputs. In the example, the first two lines are grouped together which allows to factor the sum of the corresponding input product terms as $\overline{x}_0x_1x_2 \oplus x_0\overline{x_1}x_2 = (\overline{x_0}x_1 \oplus x_0\overline{x_1})x_2$. The factor $(\overline{x_0}x_1 \oplus x_0\overline{x_1})$ is then simplified to $x_0 \oplus x_1$ and computed into a new ancilla qubit which is then afterwards uncomputed. Overall, the T-gate complexity of the resulting circuit is small, however, the total number of qubits used has increased by 1. The circuits in (b) and (c) were rendered using QCViewer [21].

4.2 Optimizations for Boolean Circuits

In general, given a set of AND expressions that are combined using OR we want to find sets of mutually exclusive statements that minimize the use of AND. We consider each AND expression to be a vertex on a graph and add edges between vertices that are mutually exclusive. Now we cover this graph using the smallest possible number of cliques using an algorithm that solves the CLIQUE-COVER problem, which asks to partition the vertices of a graph into cliques. NP-completeness of CLIQUE-COVER for given upper bound k of allowed cliques is well-known, however, practical approximation algorithms exist [8].

After finding a cliques partition each set of mutually exclusive statements can be implemented by evaluating the AND statements and combining all of the values on a single ancilla using XOR for each clique. These results can then be combined using OR statements. We can pre-process the given file in such a way that the cliques will be grouped in the output. This yields a new file, however, the effect of the reordering is that instead of OR functions now the much cheaper XOR functions can be used.

We ran REVS against a suite of benchmarks from the RevLib database. In Table 1 we report on improvements over the best known circuits. Our tool improved so far only one instance of the Maslov database, namely the benchmark that consists of computing a Boolean function that computes the bits of the permanent of a given 3×3 binary matrix. Shown in Table 1 are the qubit and gate costs for the eager cleanup method and in comparison the corresponding cost with the best circuit from the database. The total number of qubits in the first data column is the number of ancillas from the second data column plus

Table 1. Performance of REVS on a selection of benchmark circuits.

Name	Our Method			RevLib		Comparison (rel.)		Time
	Tot. Bits	Ancillas	Toffolis	Tot. Bits	Toffolis	Tot. Bits	Toffolis	
4mod5	7	2	1	7	4	1.00	0.25	0.00s
5xp1	23	6	83	23	365	1.00	0.23	0.02s
6sym	11	4	35	14	16	0.79	2.19	0.02s
alu4	61	39	2821	33	10456	1.85	0.27	3.70s
apex5	228	23	3727	1025	1860	0.22	2.00	15.59s
bw	36	3	73	87	159	0.41	0.46	0.01s
con1	13	4	16	13	63	1.00	0.25	0.01s
decod24	6	0	1	6	4	1.00	0.25	0.00s
e64	193	63	4096	195	130	0.99	31.5	0.17s
ex1010	38	18	6581	29	31219	1.31	0.21	6.92s
f51m	52	30	1774	35	6207	1.49	0.29	1.97s
frg2	336	54	8950	1219	2186	0.28	4.09	1913.09s
hwb9	33	15	2915	170	394	0.19	7.40	3.13s
max46	20	10	195	17	689	1.18	0.28	0.20s
mini-alu	9	3	14	10	10	0.90	1.40	0.00s
pdc	102	46	3222	619	1105	0.16	2.91	85.16s
rd84	26	14	170	34	50	0.76	3.40	0.13s
seq	107	31	3310	1617	3343	0.07	0.99	1.21s
spla	95	33	3232	489	1054	0.19	3.07	75.11s
sqrt8	18	6	32	18	158	1.00	0.20	0.02s
squar5	16	3	36	17	155	0.94	0.23	0.01s
t481	19	2	26	20	68	0.95	0.38	0.01s

the number of inputs and outputs. Typically, a space improvement of around 4X can be observed at an increase of the number of gates by around 3X. For the benchmarks shown in bold, our tool found a circuit that is better in both, the number of bits and the total number of gates.

It should be noted also that all ancilla bits computed by our tool are returned clean whereas some of the circuits in [31] leave garbage behind which would lead to a further increase in the gate count and the number of ancillas. In case the gate counts in RevLib were given in terms of multiply controlled gates, we converted the gates into Toffoli gates using $2n-3$ Toffoli gates per n-fold controlled NOT. Among the examples we observed with large possible improvement in terms of space was frg2 where a space reduction of almost a factor 4 was achieved. This however came at a significant increase in compilation time for this specific example which was caused by a large number of same output values which led to a large number of possible groupings. In our reference implementation all possible groupings were explored and the minimum picked which lead to the outlier in compilation cost. Using a greedy strategy for the groupings, a reduction of this compilation time is possible. The compilation time are measured with respect to an Intel i7-3667 @ 2 GHz 8 GB RAM processor running on a standard laptop.

We implemented the procedure that first performs the offline conversion of the given circuit to an equivalent circuit by performing the clique-cover-based XOR maximization. Then this circuit is converted directly into an MDD before cleanup and in doing so, our compiler finds the optimized grouping that replaces OR terms with XOR terms. As the next stage in the pre-processing, we then use the Exorcism-4 tool [18] to perform Exclusive Sum-Of-Product (ESOP) minimization. Afterward, we use factoring techniques from multi-level circuit optimization and minimize the size of the out expressions. A pseudo-code implementation of this factoring technique is given as Algorithm 2.

An example of how our Boolean expression generation allows to trade circuit size (and compilation time) for the total number of qubits used is shown in Fig. 2. Finally, since the processed PLA file is an xor sum on the outputs, MDD based cleanup can be done after each boolean expression to minimize the number of bits use.

5 Conclusions

We developed a tool that automates the translation of classical, irreversible programs into reversible programs. Contrary to previous approaches of reversible programming languages such as the reversible languages R or Janus [23], our language does not constrain the programmer. Also, in contrast to previous approaches for implementing Bennett-style strategies such as Quipper [9] our approach is more space efficient. We employ heuristic strategies which seek to identify parts of the program that lead to mutation which then can be implemented via in-place operations.

In order to manage the arising data dependencies, we introduced MDD graphs which capture data dependencies as well as data mutation in one graph. We prove that our eager cleanup strategy is correct, provided the mutation paths that occur in the MDD have no inter-path dependency. In case such dependencies arise, we clean up the paths using the standard Bennett strategy, which allows us to compile any program that can be expressed in our language into a Toffoli network.

We found examples where our dependency-graph based method for eager cleanup is better than Bennett's original method, even when Bennett's method is implemented by cleaning up at function boundaries. Using an example benchmark suite compiled from the classical circuits and systems community as well as known reversible benchmarks, we show that the method can be applied for medium to large scale problems.

References

1. Revlib - an online resource for reversible functions and circuits. http://www.revlib.org/
2. Aho, A.V., Lam, M.S., Sethi, R., Ullman, J.D.: Compilers: Principles, Techniques, and Tools. Addison Wesley, London (2007)
3. Bennett, C.H.: Logical reversibility of computation. IBM J. Res. Dev. **17**, 525–532 (1973)
4. Bennett, C.H.: Time/space trade-offs for reversible computation. SIAM J. Comput. **18**, 766–776 (1989)
5. Buhrman, H., Tromp, J., Vitányi, P.: Time and space bounds for reversible simulation. In: Orejas, F., Spirakis, P.G., Leeuwen, J. (eds.) ICALP 2001. LNCS, vol. 2076, pp. 1017–1027. Springer, Heidelberg (2001). doi:10.1007/3-540-48224-5_82
6. Pebble games and complexity. Ph.D. thesis, Electrical Engineering and Computer Science, UC Berkeley, Technical report: EECS-2013-145 (2013)
7. Chattopadhyay, A., Pal, N., Majumder, S.: Ancilla-quantum cost trade-off during reversible logic synthesis using exclusive sum-of-products (2014). arxiv:1405.6073
8. Goldschmidt, O., Hochbaum, D.S., Hurkens, C.A.J., Yu, G.: Approximation algorithms for the k-clique covering problem. SIAM J. Disc. Math. **9**(3), 492–509 (1996)
9. Green, A., LeFanu Lumsdaine, P., Ross, N., Selinger, P., Valiron, B.: Quipper: a scalable quantum programming language. In: PLDI 2013 (2013)
10. Heckey, J., Patil, S., Javadi Abhari, A., Holmes, A., Kudrow, D., Brown, K.R., Franklin, D., Chong, F.T., Martonosi, M.: Compiler management of communication and parallelism for quantum computation. In: ASPLOS 2015, pp. 445–456. ACM (2015)
11. JavadiAbhari, A., Patil, S., Kudrow, D., Heckey, J., Lvov, A., Chong, F.T., Martonosi, M.: ScaffCC: scalable compilation and analysis of quantum programs. Parallel Comput. **45**, 2–17 (2015)
12. Lange, K.J., McKenzie, P., Tapp, A.: Reversible space equals deterministic space. J. Comput. Syst. Sci. **60**(2), 354–367 (2000)
13. Lin, C.-C., Jha, N.K.: RMDDS: Reed-Muller decision diagram synthesis of reversible logic circuits. ACM J. Emerg. Technol. Comput. Syst. **10**(2), 14 (2014)
14. Maslov, D.: Reversible logic synthesis benchmarks page. http://webhome.cs.uvic.ca/~dmaslov/
15. Maslov, D., Miller, D.M., Dueck, G.W.: Techniques for the synthesis of reversible Toffoli networks. ACM Trans. Des. Autom. Electron. Syst. **12**(4), 42 (2007)
16. Minkovich, K.: BLIF benchmark suite. http://cadlab.cs.ucla.edu/~kirill/
17. Mishchenko, A., Brayton, R., Chatterjee, S.: Boolean factoring and decomposition of logic networks. In: Proceedings of the IEEE/ACM International Conference on Computer-Aided Design, pp. 38–44. IEEE Press (2008)
18. Mishchenko, A., Perkowski, M.: Fast heuristic minimization of exclusive sum-of-products, 2001. Exorcism is available as part of the ABC software. https://people.eecs.berkeley.edu/~alanmi/
19. Muchnick, S.S.: Compiler Design and Implementation. Morgan Kaufmann, San Francisco (1997)
20. Nielsen, M.A., Chuang, I.L.: Quantum Computation and Quantum Information. Cambridge University Press, Cambridge (2000)
21. Parent, A., Parker, J., Burns, M., Maslov, D.: Quantum Circuit Viewer. Poster presentation at TQC 2013, University of Guelph, Canada. Software (2013). https://github.com/aparent/QCViewer, http://qcirc.iqc.uwaterloo.ca/

22. Parent, A., Roetteler, M., Svore, K.M.: Reversible circuit compilation with space constraints (2015). arXiv:1510.00377
23. Perumalla, K.S.: Introduction to Reversible Computing. CRC Press, Boca Raton (2014)
24. Saeedi, M., Markov, I.L.: Constant-optimized quantum circuits for modular multiplication and exponentiation. Quantum Information and Computation **12**(5&6), 361–394 (2012)
25. Saeedi, M., Markov, I.L.: Synthesis and optimization of reversible circuits - a survey. ACM Comput. Surv. **45**(2), 21 (2013)
26. Shafaei, A., Saeedi, M., Pedram, M.: Reversible logic synthesis of k-input, m-output lookup tables. In: DATE 2013, pp. 1235–1240 (2013)
27. Soeken, M., Robert Wille, R., Hilken, Ch., Przigoda, N., Drechsler, R.: Synthesis of reversible circuits with minimal lines for large functions. In: Proceedings of ASP-DAC 2012 (2012)
28. Syme, D., Granicz, A., Cisternino, A.: Expert F# 3.0. Apress Publishing, New York (2012)
29. Thomsen, M.K.: A functional language for describing reversible logic. In: Forum on Specification and Design Languages, pp. 135–142. IEEE (2012)
30. Viamontes, G.F., Markov, I.L., Hayes, J.P.: Quantum Circuit Simulation. Springer, Heidelberg (2009)
31. Wille, R., Drechsler, R.: BDD-based synthesis of reversible logic for large functions. In: Proceedings of DAC 2009, pp. 270–275 (2009)
32. Wille, R., Drechsler, R.: Towards a Design Flow for Reversible Logic. Springer, Dodrecht (2010)
33. Wille, R., Offermann, S., Drechsler, R.: SyReC: a programming language for synthesis of reversible circuits. In: Specification Design Languages (FDL), pp. 1–6 (2010)
34. Wille, R., Soeken, M., Drechsler, R.: Reducing the number of lines in reversible circuits. In: Proceedings of DAC 2010, pp. 647–652 (2010)
35. Wille, R., Soeken, M., Miller, D.M., Drechsler, R.: Trading off circuit lines and gate costs in the synthesis of reversible logic. Integration **47**(2), 284–294 (2014)
36. Yokoyama, T., Glück, R.: A reversible programming language and its invertible self-interpreter. In: PEPM 2007, pp. 144–153 (2007)