Unlocking Efficiency and Scalability of Reversible Logic Synthesis using Conventional Logic Synthesis

Mathias Soeken
EPFL, Switzerland
mathias.soeken@epfl.ch

Anupam Chattopadhyay
Nanyang Technological University, Singapore
anupam@ntu.edu.sg

ABSTRACT

Latest quantum technologies promise realization of extremely large circuits, whereas, reversible logic synthesis, the key automation step for quantum computing, suffers from scalability bottleneck. Scalability can be achieved with Decision Diagram (DD)-based synthesis at the cost of significant ancilla/garbage lines overhead. In this paper, we present a novel hierarchical reversible logic synthesis, where DD-based synthesis is invoked within an And-Inverter Graph (AIG)-based synthesis wrapper, balancing scalability and performance.

The resulting tool can synthesize much larger functions (512-inputs), provides excellent flexibility, and restricts ancilla overhead. On average, line-count and gate-count reductions of 94% and 35% respectively, are achieved, compared to state-of-the-art.

1. INTRODUCTION

Reversible logic synthesis is fast emerging as major research direction for enabling the realization and deployment of future technologies such as quantum computing. In contrast to the conventional logic synthesis that utilizes the universal operator-set of irreversible logic gate, e.g., NAND gate, quantum computing mandates that the underlying logical operations are inherently reversible [21]. As a result, logical primitives with reversible form, e.g., NOT, CNOT, and Toffoli [4] are used for realizing a quantum circuit. The process of mapping a given Boolean multi-output function $f : \{0,1\}^n \to \{0,1\}^m$ to a set of reversible logic gates is known as reversible logic synthesis. Reversible logic synthesis begins from a given Boolean function, which can be irreversible. The first step, in that case, is to convert it to a reversible Boolean function which possibly requires additional lines. When additional constant-initialized input Boolean variables are needed for constructing the output function, those are referred to as *ancilla* inputs. If an ancilla input is not recovered to its initial value after the computation is done, it is referred to as a *garbage* output. For practical implementation purposes, it is desirable to keep the ancilla count as small as possible. Determining the minimum number of required ancilla bits is coNP-complete [29]. Reversible logic synthesis approaches that guarantee a minimum number of ancilla lines are called *ancilla-free*. Though being far from practical day-to-day usage, quantum computing is making rapid strides with recent results reporting quantum gate realizations using silicon [30] and novel architectures allowing scalability to millions of physical qubits [13]. A range of quantum algorithms are known to have polynomial, superpolynomial, or exponential speed-up over classical algorithms.[1] A key constituent of these algorithms are large Boolean functions, such as, modular exponentiation function in Shor's factoring algorithm [25]. Automatic synthesis of such large Boolean functions is a major, unsolved research problem. This is only partially addressed by some dedicated, circuit-specific synthesis flows [14, 10].

DAC '16, June 05-09, 2016, Austin, TX, USA

DOI: http://dx.doi.org/10.1145/2897937.2898107

1.1 Reversible Logic Synthesis: Structural or Functional

In the last few years, with the emergence of novel technology devices, logic synthesis also made a prominent comeback from industry-standard tools to an important research problem. In particular, researchers have tried to closely match the intermediate representation (IR) of a logic function with the elementary gates that are expressed in a given technology. State-of-the-art examples of these are Or/And-inverter graphs [19], Majority-inverter graphs [3], Bi-conditional binary decision diagrams [2, 9], and threshold logic networks [33]. Reversible logic synthesis is no exception, where, in addition to the adaptation of the conventional logic synthesis approaches, novel IRs such as QMDDs [18], QuiDDs [31], and RbDDs [1] are proposed.

Based on the IR, we draw the attention to another classification of the logic synthesis approaches, which we refer to as *functional* and *structural* approach.

- *Functional logic synthesis approach:* Functional logic synthesis is enabled if the IR is used to explicitly express the logic function. Examples for IRs are Boolean truth tables or binary decision diagrams (BDDs [6]).
- *Structural logic synthesis approach:* Structural logic synthesis is enabled if the IR is used to represent the structure of the circuit, e.g., using And-inverter graphs (AIGs). Such an IR provides the capability to perform structural manipulation during the synthesis, e.g., algebraic factorization. However, to evaluate the overall function value, even for a sub-part of the IR, requires exhaustive enumeration of all the input values.

Both types of logic synthesis approaches come with advantages and drawbacks. Only structural logic synthesis is scalable to very large functions, however, only functional logic synthesis can guarantee optimality w.r.t. some cost metric. In order to take the *best of both worlds*, one can

[1] http://math.nist.gov/quantum/zoo

combine both strategies into one hybrid synthesis approach. The overall approach is structural but employs functional logic synthesis for substructures of the IR.

1.2 Related Work

During the last decade, a significant amount of research has been done in the field of reversible logic synthesis. These works can be broadly classified into two classes. First, there are optimal/pseudo-optimal approaches based on techniques such as satisfiability solving [12], reachability analysis, and exhaustive enumeration [11]. Second, there are a suite of heuristic algorithms, which are inspired from the conventional logic synthesis techniques. For this paper, we concentrate on the latter class of algorithms and kindly refer inquisitive reader to the excellent survey available at [23].

The heuristic algorithms either adopt structural (e.g., using BDDs [32, 15] when regarded as a MUX circuit) or functional synthesis approach (e.g., based on truth tables [17] or also on BDDs [28, 27] when regarded as symbolic function representation). In order to guarantee minimum ancilla count for a reversible circuit implementation, it is necessary to have a full functional view of the Boolean function. This is only accessible in functional synthesis approaches. In fact, since structural synthesis approaches need to store intermediate results on temporary lines, today's state-of-the-art structural synthesis approaches add a tremendous amount of additional lines [32, 15]—magnitudes larger than the theoretical upper bound [29]. Despite having the possibility to perform functional evaluation and reduce ancilla count, the DD-based reversible logic synthesis methods, initially, adopted a node-wise structural synthesis approach. For each node, a corresponding reversible circuit is constructed and stitched with the outputs of the descendant nodes to derive the entire circuit. Recently, it was shown that by utilizing a fast symbolic simulation [27], ancilla-free reversible logic synthesis for large Boolean functions is possible. However, it was also noted in [27], that for complex functions with large number of inputs, the runtime increases exponentially.

1.3 Motivation and Contribution

So far, to the best of our knowledge, there is no reversible logic synthesis that merged functional and structural techniques in the same synthesis flow, whereas, this is not uncommon in conventional logic synthesis [8]. A recent work outlining a mixed, hierarchical reversible logic synthesis [16] proposes the global synthesis to be driven by Reed-Muller synthesis and local synthesis with decision diagrams. However, efficient Reed-Muller synthesis is achieved by DD-based heuristics [20], which, in effect, limits the scalability. Indeed, the hierarchical approach outlined in [16] does not report results for benchmarks larger than 19 inputs.

This work presents a new hybrid logic synthesis approach for reversible circuits. The structural IR to represent the overall function are AIGs which are known to be more scalable compared to BDDs. During synthesis, sub-circuits are extracted from the AIG that are applicable to ancilla-free synthesis approaches thereby locally guaranteeing circuits with the minimum number of lines. The major challenges associated with this approach are i) enabling a user-controllable threshold for the decision of structural and functional synthesis flows; ii) performing fast, ancilla-free synthesis by transforming the representation from AIG to DD; and iii) stitching the sub-circuits together without introducing any redundancy. We successfully addressed all these challenges as described in detail in Section 2.

Our proposal is inspired from technology mapping methods

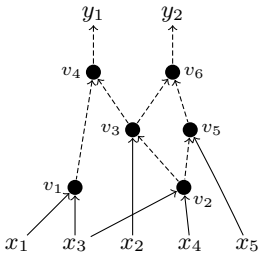
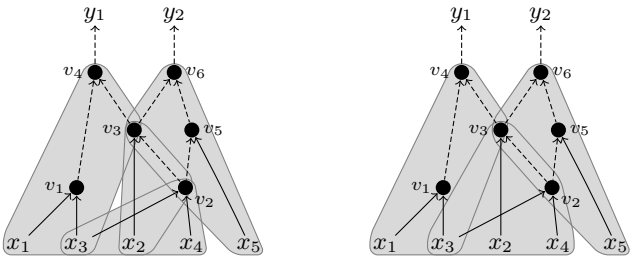


Figure 1: And-inverter graph



(a) Fanout-free regions (b) Synthesizable regions

Figure 2: Example application of Algorithm C, steps 2–3

used in conventional logic synthesis flows [22, 8, 7]. Experimental evaluations show that our proposed approach outperforms the state-of-the-art reversible logic synthesis flows in several aspects:

1. *Quality of the result:* Experiments show that gate count is reduced in most of the cases and line count can be reduced by more than three orders of magnitude.
2. *Flexibility:* Due to the canonicity property of BDDs, hybrid BDD is less flexible as it allows fewer structural choices. Since AIGs are not canonical, they allow for a higher flexibility, e.g., due to the possibility of rewriting. This in turn allows for a variety of different synthesis results enabling design space exploration in the design flow.
3. *Scalability:* AIGs are more scalable than BDDs and allow the application to functions that couldn't be synthesized before. Besides that, algorithmic specific parameters in our approach allow to trade-off quality of the synthesis result with run-time.

2. ALGORITHM

2.1 General Idea

The main idea of our proposed hybrid synthesis approach is to represent the function as a circuit logic representation, in our case as AND-inverter graph. Such representations are scalable, particularly compared to BDDs. The algorithm partitions the circuit into smaller sub-circuits that are as large as possible in order to be applied for ancilla-free reversible logic synthesis. This allows to keep the overall number of additional lines small. Recent advances in symbolic ancilla-free synthesis approaches further allow to increase the size of such sub-circuits. Since reversible functions have the same number of input variables and output variables, these sub-circuit should be determined such that they ideally have the same number of inputs and outputs as well.

In our approach, we first partition the circuit into *fanout-free regions* (FFR) of bounded input width and then merge such fanout-free regions into so-called *synthesizable regions*. An FFR (r, L) rooted in a gate r is a connected sub-circuit with leaves L such that all internal gates $g \notin L \cup \{r\}$ only fan out to one other gate, which must by definition also be in the sub-circuit. An FFR has only one output variable and therefore is not ideal for reversible logic synthesis. Merging several FFRs into a synthesizable region increases the number of outputs. When merging two or more FFRs, common input variables are shared.

This two step-approach is helpful in controlling the size of the resulting sub-circuits and ensuring to keep the number of internal vertices with fan-out small. Vertices with fan-out need to be preserved in a reversible circuit by adding lines, as reversible circuits prohibit fan-out.

2.2 Implementation

This section first gives a high-level description of the algorithm and then explains each step in detail along with a running example based on Fig. 1. It shows an AIG that represents the ISCAS function '*c17*'. Filled circles represent AND gates and dashed lines are complemented edges.

Algorithm C (*Hybrid circuit-based synthesis*). This is a high-level description of our algorithm that takes a circuit C (in our case represented as an AIG) and returns a reversible circuit R which realizes an embedding of the function represented by C. One threshold parameter t controls the size of the sub-circuits and can be used to trade-off the quality of the resulting circuit with the overall run-time.

C1. [Add inputs.] For each input in C add a circuit line to R.

C2. [Compute FFRs.] Compute fanout-free regions

$$(r_1, L_1), \ldots, (r_\ell, L_\ell)$$

in C such that $r_i \prec r_{i+1}$ for all $1 \leq i < \ell$ and $|L_i| \leq \frac{t}{2}$ for all $1 \leq i \leq \ell$. ('$\prec$' denotes topological order).

C3. [Merge FFRs.] Merge successive fanout-free regions into synthesizable regions

$$(I_1, O_1, P_1), \ldots, (I_k, O_k, P_k)$$

such that $|I_i| + |O_i| \leq t$ for all $1 \leq i \leq k$.

C4. [Synthesis.] For $i = 1, \ldots, k$, synthesize the synthesizable region (I_i, O_i, P_i) and append it to the reversible circuit R.

C5. [Add outputs.] Add outputs of C to R.

In the following, each step of Algorithm C is described in more detail.

C1. Add inputs

In the initialization of the algorithm we add one circuit line to R for each primary input in C. A primary input in C is an AIG node. In general, Algorithm C keeps track of which AIG node v is currently represented by which circuit line in R by regularly updating a map `LINE[`v`]`. Initially `LINE` maps each primary input to the corresponding added circuit line.

EXAMPLE 1. *This and the following examples demonstrate the application of Algorithm C to the AIG in Fig. 1 with t set to 6. After step C1, the reversible circuit R consists of 5 empty circuit lines and we have*

$$\texttt{LINE[}x_1\texttt{]} = 0, \quad \texttt{LINE[}x_2\texttt{]} = 1, \quad \texttt{LINE[}x_3\texttt{]} = 2,$$
$$\texttt{LINE[}x_4\texttt{]} = 3, \quad \texttt{LINE[}x_5\texttt{]} = 4.$$

C2. Compute FFRs

In many logic synthesis applications one is typically interested in the maximum FFRs, which can be computed using a depth-first traversal that stops at nodes that are either primary inputs or have more than one fanout. However, in our application we are interested in FFRs that have a bounded number of leaves. Therefore, the FFR must be computed using a breadth-first traversal which extends the leaf boundary in every step and stops as soon the given threshold is reached.

Step 2 also requires that FFRs are ordered such that their roots are in topological order. This is achieved by computing FFRs starting from the primary outputs. For this purpose, we keep track of a root boundary R, which is initially a sequence of all primary outputs in topological order. In each step we take and remove the largest element r of R, compute a bounded FFR (r, L), and then insert all elements of L into R such that its elements remain in topological order. This procedure ensures that all FFRs are computed in reverse topological order.

EXAMPLE 2. *Fig. 2(a) shows all FFRs of the circuit in Fig. 1:*

$$(v_2, \{x_2, x_3, x_4\}), (v_3, \{x_2, v_2\}),$$
$$(v_4, \{x_1, x_3, v_3\}), (v_6, \{x_5, v_2, v_3\})$$

Their roots are in topological order, i.e., $v_2 \prec v_3 \prec v_4 \prec v_6$.

C3. Merge FFRs

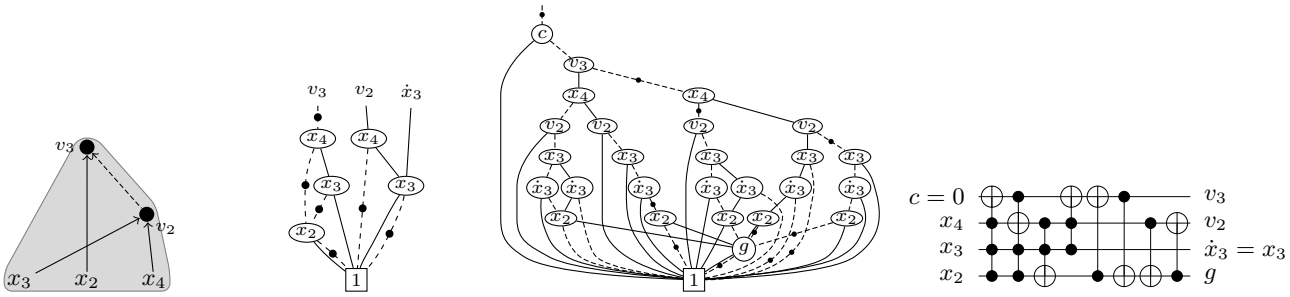
Each FFR (r, L) represents a single-output Boolean function over $|L|$ variables. Synthesizing f as a reversible function requires $|L| + 1$ lines, unless f is balanced [29], which gets less likely when $|L|$ increases. In order to save additional lines during synthesis we are interested in sub-circuits that have more than one output. For this purpose, we merge successive FFRs into synthesizable regions (I, O, P) with inputs I and outputs O. Since reversible circuits prohibit fanout, the circuit inputs cannot be accessed after being updated by a gate. Some inputs of I may be included in other synthesizable regions and therefore they need to be explicitly preserved by exposing them to the outputs. These inputs are stored in $P \subseteq I$. One can use reference counting in order to track which of the inputs need to be preserved.

The function that is represented by the synthesizable region (I, O, P) has $|I|$ inputs and $|O| + |P|$ outputs. A reversible circuit that realizes this function requires at most $|I| + |O|$ variables. ESOP-based synthesis approaches such as [24] imply this bound, since they require one line per input and one line per output, also all inputs are preserved. The sum of inputs I and outputs O is used to control the size of the resulting synthesizable regions using the threshold t.

EXAMPLE 3. *Remember that t is set to 6 in this example. FFRs are merged in successive order, i.e., first one tries to merge $(v_2, \{x_2, x_3, x_4\})$ and $(v_3, \{x_2, v_2\})$ which results in a synthesizable region*

$$(I = \{x_2, x_3, x_4\}, O = \{v_2, v_3\}, P = \{x_3\}).$$

The input x_3 is part of the FFR that is rooted in v_4 and therefore needs to be preserved. Notice that also v_2 is still required to realize the FFR that is rooted in v_6 and therefore needs to be added to the outputs. With a size of 6 this synthesizable region respects the threshold, however, merging the next FFR would exceed it. For the same reason the remaining two FFRs rooted in v_4 and v_6 cannot be merged



(a) Synthesizable region (b) Symbolic simulation (c) Embedding (d) Circuit

Figure 3: Example application of Algorithm C, step 4

and are directly taken as synthesizable regions

$$(\{x_1, x_3, v_3\}, \{v_4\}, \{v_3\}) \text{ and } (\{x_5, v_2, v_3\}, \{v_6\}, \{\}).$$

As v_3 is used as input in both these synthesizable regions, it needs to be preserved by the first one.

C4. Synthesis

Each synthesizable region represents an irreversible function f to be synthesized using a functional synthesis approach. First, the synthesizable region is translated into a BDD using symbolic simulation. Symbolic simulation is key in every hybrid synthesis approach to translate structural parts into functional representations. Based on this BDD the minimal number of additional inputs can be computed using the technique proposed in [29]. This technique requires the sum-of-product representation of f as input which may be costly to generate from the BDD in memory. We adopted this algorithm for embedding the BDD and modified it such that it works directly on the BDD representation. The result is a BDD that represents the characteristic function of a reversible function that embeds f. This BDD can be used to perform synthesis using an ancilla-free synthesis algorithm such as proposed in [27]. If the synthesizable region is small enough, the truth table of f can be computed and any truth table based synthesis approach can be applied to obtain the reversible sub-circuit. The hierarchical technique also opens up the possibility to execute multiple, parallel runs for the synthesizable regions and selecting the one with the best performance.

Example 4. *Fig. 3 shows how a realization is obtained for the first synthesizable region (Fig. 3(a)). Symbolic simulation leads to the BDD shown in Fig. 3(b). The output $\dot{x}_3$ corresponds to the preserved input x_3. From this BDD the minimal number of additional lines is determined to be* 1 *using the algorithm from [29]. Embedding then leads to a BDD depicted in Fig. 3(c) which represents a (partially specified) reversible function that embeds the function realized by the synthesizable region. Its additional constant input is c and the additional garbage output is g. Using a symbolic ancilla-free synthesis approach such as [27] leads to a circuit as depicted in Fig. 3(d).*

We perform the following optimization to reduce run-time. Whenever $I = P$ in the synthesizable region, i.e., all inputs must be preserved. In this case, the number of lines in the reversible circuit must be at least $|I| + |O|$ and we can apply ESOP-based reversible logic synthesis [24].

After the reversible sub-circuit has been determined it needs to be appended to the overall reversible circuit R. For each constant line in the sub-circuit a new line is added. For the remaining lines, the map `LINE` can be used to arrange the gates. Afterwards, the line mapping is updated according to the new outputs. Garbage outputs can be used to remove previous line mappings.

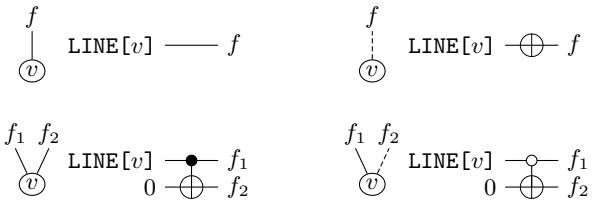


Figure 4: Add outputs

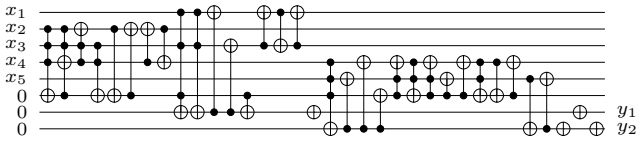


Figure 5: Resulting circuit

Example 5. *In the running example, a sixth line with index 5 is added to R for the constant input. After appending the sub-circuit in Fig. 3(d) to R, the line mapping is updated:*

$$\texttt{LINE}[v_3] = 5, \quad \texttt{LINE}[v_2] = 3$$

Note that `LINE`$[x_3]$ *remains unchanged and x_2 is removed from the mapping.*

C5. Add outputs

An output in the AIG is a pointer to a node v which may be complemented. The line mapping `LINE`$[v]$ then reveals the circuit line of that output. In case the output is complemented, a NOT gate is appended to that line. It may be possible that several outputs point to the same node. However, in reversible circuits no fanout is allowed and therefore each output needs its own line. For this purpose we store for each first considered output the line it maps to. If another output points to the same node, an additional line is added to the reversible circuit and the value of the output is copied using a CNOT gate. If the polarities of the outputs differ, a CNOT with a negative control line is added instead.

EXAMPLE 6. *Fig. 4 displays all possible scenarios of how AIG outputs are handled in the reversible circuit.*

The overall reversible circuit for the AIG in Fig. 1 with a threshold $t = 6$ is depicted in Fig. 5.

3. EXPERIMENTAL EVALUATION

We have implemented the proposed approach in C++ on top of RevKit [26] in the command `cbs`.[2] We have performed two experiments: (i) a comparison to the state-of-the-art and an evaluation of the trade-off parameter and (ii) an evaluation of the effect of AIG optimization to the final synthesis results. All experiments have been carried out on an 2.6 GHz Intel Xeon CPU with 128 GB RAM running Linux 3.13.

3.1 Comparison and Trade-off

The state-of-the-art structural synthesis approach that is available through open source software is the hierarchical BDD based synthesis [32] and therefore we have used it as baseline for comparison to our proposed approach. We have not applied any post-synthesis optimization approaches as they can be applied to both approaches. As benchmark set we used the ISCAS benchmarks. In order to apply the BDD based synthesis approach to these benchmarks, we generate the BDDs using symbolic simulation on the AIG without first translating them into a sum-of-product representation. This is an extension compared to the previously proposed approach.

Table 1 list the results of the experimental evaluation. It shows the benchmark with its number of inputs and outputs (I/O). For the hierarchical BDD based approach and our proposed approach with thresholds $t \in \{9, 11, 13, 15\}$ the table lists number of gates (d), number of lines (l), and the run-time. We compute the improvement in additional lines and gate count compared to the BDD based synthesis approach and give the average over all improvements in the last row of the table.

First it can be noticed that the number of additional lines can be reduced tremendously, on average the improvement is between 94% and 96% depending on the threshold. The number of gates is not necessarily reduced and increases when the threshold is set larger. This is in agreement with many previous observations in reversible logic and quantum computation. Note that, circuit implementation with low ancilla count is of significant importance due to the direct implication on the number of qubits, and ancilla-free synthesis with truth table or BDD does not scale for functions large variable, as discussed earlier. Also the run-times increase when the threshold is increased, however, for a small t the run-times are comparable to the BDD based synthesis. For the benchmark '*c6288*', a multiplier, the BDD cannot be constructed and hence the BDD based synthesis approach cannot be applied. However, using a threshold of 9 our proposed approach can find a solution within 5 seconds. The effect of scalability is more prominent, in the next set of experiments with large combinational benchmarks.

The table shows no quantum cost information as there are several technologies which are based on different assumptions. Often, additional lines have a positive effect on the technology dependent quantum costs since functionality can be implemented using gates with a smaller number of control lines. As such the generated circuits can be considered as starting point for quantum circuit synthesis in which the reversible gates are decomposed into smaller ones.

[2]http://github.com/msoeken/cirkit

3.2 Effect of AIG Optimization

Since our proposed hybrid approach works structurally on the AIG representation, different AIGs for the same function lead to different synthesis results. There exists a vast amount of optimization approaches for AIGs, which can be directly used to obtain alternative synthesis results. This section describes an experiment that evaluates the effect of optimization. We have applied our proposed approach to the arithmetic instances of the EPFL combinational benchmark suite.[3] These are significantly larger than the ISCAS benchmarks and for all of them the BDD construction using symbolic simulation on the input AIG could not be completed within the provided computing and memory resources.

The results are listed in Table 2. We applied the hybrid synthesis approach with $t = 10$ to three different AIGs: (i) the original AIG provided from the benchmark suite, (ii) the optimized AIG using ABC's `if -x -g` command [5], and (iii) the optimized AIG using ABC's `dc2` command. The first command priotorizes depth and the second command priotorizes size in the optimization. The table lists size and depth of the corresponding AIG, number of gates (d) and number of lines (l) of the reversible circuit, as well as run-time required for synthesis.

The experiment shows that the AIG representation can have a significant impact on the outcome of the synthesis result, e.g., '*bar*' and '*div*' in case of the `dc2` optimization. Optimization may not necessarily lead to better results, as for example shown by many instances of the `if -x -g` optimization. Hence, tailored AIG optimization approaches aiming for a better synthesis result are of great interest.

4. CONCLUSIONS

Reversible logic synthesis research have made significant progress over the last few years, however, it still suffers from scalability. Conventional logic synthesis tools adopt a structural approach towards synthesis, which scales better. Structural model of synthesis in reversible domain leads to high number of ancilla lines, which is not desirable for practical quantum circuits. In this paper, we propose a merger of structural and functional logic synthesis to address both concerns. The proposed hierarchical logic synthesis resorts to an AIG structure globally, with local calls to functional synthesis flows. The performance of this hierarchical flow conclusively demonstrates a significant improvement in scalability, tremendous reduction in ancilla overhead and flexibility (via threshold) to control the ancilla vs gate count performance trade-off. As a bonus, we observe interesting correlations between the AIG optimizations and final circuit performance figures. This can lead to a detailed study in future. Further future works include, multiple, parallel synthesis runs for the local optimizations, including known ancilla-free methods like transformation-based, BDD-based and optimal synthesis methods.

Acknowledgments. This research was partly supported by H2020-ERC-2014-ADG 669354 CyberCare and by the European COST Action IC 1405 'Reversible Computation'.

5. REFERENCES

[1] A. Abdollahi, M. Saeedi, and M. Pedram. Reversible logic synthesis by quantum rotation gates. *QIC*, 13(9–10):771–702, 2013.

[2] L. Amarú, P.-E. Gaillardon, and G. De Micheli. Biconditional BDD: A novel canonical BDD for logic synthesis targeting XOR-rich circuits. In *DATE*, pages 1014–1017, 2013.

[3]http://lsi.epfl.ch/benchmarks

Table 1: Evaluating different thresholds and comparing to hybrid BDD approach using ISCAS benchmarks

		hierarchical BDD [32]			$t = 9$			$t = 11$			$t = 13$			$t = 15$		
Benchmark	I/O	d	l	time	d	l	time	d	l	time	d	l	time	d	l	time
c432	36/7	57805	14944	0.42	6389	93	1.53	11850	85	10.73	46024	83	229.33	175888	83	9851.06
c499	41/32	113135	23503	0.76	2798	99	0.52	8632	96	2.20	48175	100	66.58	74069	93	160.40
c880	60/26	14043	3771	6.35	12263	184	3.50	41133	170	47.78	184073	155	815.66	438909	151	10640.20
c1355	41/32	113167	23503	0.80	5452	193	1.01	10282	120	2.07	37301	135	33.12	103846	120	324.89
c1908	33/25	26822	5590	0.30	3322	107	0.66	7736	103	2.60	35546	96	43.57	64811	94	120.21
c2670	157/64	49567	11988	104.67	11159	384	2.02	29803	325	16.53	59251	276	114.81	267367	277	5142.16
c3540	50/22	323309	78195	47.05	23053	382	8.22	76771	326	63.51	258500	318	807.46	1023729	308	24749.80
c5315	178/123	7627	1973	0.30	28359	768	7.61	67691	576	34.75	97978	455	120.35	340986	447	1751.77
c6288	32/32			MO	46805	861	4.76	117334	850	23.12	336451	748	247.93	1172413	682	3025.67
Avg. impr.					35%	94%		-66%	95%		-285%	96%		-1044%	96%	

Table 2: Evaluating the effect of AIG optimization using EPFL benchmarks

		Original version					Optimized with `if -x -g`					Optimized with `dc2`				
Benchmark	I/O	size	depth	d	l	time	size	depth	d	l	time	size	depth	d	l	time
adder	256/129	1020	255	28066	511	2.58	1235	171	35963	599	4.52	1019	255	25352	509	2.44
bar	135/128	3336	12	50045	807	47.78	3141	12	48710	713	31.35	2952	14	44629	591	24.75
div	128/128	57247	4372	1684340	28439	448.47	64380	2961	1489691	24229	578.53	40520	4403	807564	20000	146.37
hyp	256/128	214335	24801	7021956	91144	2668.29	244233	16660	5832247	92519	3074.16	211689	24882	6990587	87654	2578.97
log2	32/32	32060	444	716191	8779	419.17	33005	278	669769	8621	417.88	29323	358	348692	8680	217.70
max	512/130	2865	287	96479	1027	77.47	2975	177	112098	1114	98.65	2831	211	99905	1010	101.52
multiplier	128/128	27062	274	299537	5420	183.98	27775	186	412347	6351	196.12	24377	262	248257	5647	108.87
sin	24/25	5416	225	145367	1689	151.33	5802	136	154582	1836	150.99	5040	161	91750	1692	117.34
sqrt	128/64	24618	5058	1096263	6714	260.94	27101	4095	692258	6677	140.62	18453	6070	519703	5727	86.27
square	64/128	18484	250	415976	6351	114.88	18786	168	493280	6685	196.29	16674	247	508744	7174	158.36

[3] L. Amarú, P.-E. Gaillardon, and G. De Micheli. Majority-inverter graph: A novel data-structure and algorithms for efficient logic optimization. In *DAC*, pages 194:1–194:6, 2014.

[4] A. Barenco, C. H. Bennett, R. Cleve, D. P. DiVincenzo, N. Margolus, P. Shor, T. Sleator, J. A. Smolin, and H. Weinfurter. Elementary gates for quantum computation. *Phys. Rev. A*, 52:3457–3467, Nov 1995.

[5] R. K. Brayton and A. Mishchenko. ABC: an academic industrial-strength verification tool. In *CAV*, pages 24–40, 2010.

[6] R. E. Bryant. Graph-based algorithms for Boolean function manipulation. *IEEE T-C*, C-35(8):677–691, 1986.

[7] S. Chatterjee, A. Mishchenko, and R. Brayton. Factor cuts. In *ICCAD*, pages 143–150, 2006.

[8] S. Chatterjee, A. Mishchenko, R. K. Brayton, X. Wang, and T. Kam. Reducing structural bias in technology mapping. *IEEE T-CAD*, 25(12):2894–2903, 2006.

[9] A. Chattopadhyay, A. Littarru, L. Amaru, P.-E. Gaillardon, and G. De Micheli. Reversible logic synthesis via biconditional binary decision diagrams. In *ISMVL*, pages 2–7, 2015.

[10] A. Chattopadhyay, S. Majumder, C. Chandak, and N. Chowdhury. Constructive reversible logic synthesis for Boolean functions with special properties. In *RC*, pages 95–110. 2014.

[11] O. Golubitsky and D. Maslov. A study of optimal 4-bit reversible Toffoli circuits and their synthesis. *IEEE T-C*, 61(9):1341–1353, 2012.

[12] D. Große, R. Wille, G. W. Dueck, and R. Drechsler. Exact multiple-control Toffoli network synthesis with SAT techniques. *IEEE T-CAD*, 28(5):703–715, 2009.

[13] C. D. Hill, E. Peretz, S. J. Hile, M. G. House, M. Fuechsle, S. Rogge, M. Y. Simmons, and L. C. L. Hollenberg. A surface code quantum computer in silicon. *Science Advances*, 1(9), 2015.

[14] I. L. Markov and M. Saeedi. Faster quantum number factoring via circuit synthesis. *Phys. Rev. A*, 87:012310, Jan 2013.

[15] M. Krishna and A. Chattopadhyay. Efficient reversible logic synthesis via isomorphic subgraph matching. In *ISMVL*, pages 103–108, 2014.

[16] C.-C. Lin and N. K. Jha. RMDDS: Reed-Muller decision diagram synthesis of reversible logic circuits. *JETC*, 10(2):14:1–14:25, 2014.

[17] D. Maslov, G. W. Dueck, and D. M. Miller. Toffoli network synthesis with templates. *Computer-Aided Design of Integrated Circuits and Systems, IEEE Transactions on*, 24(6):807–817, June 2005.

[18] D. M. Miller and M. A. Thornton. QMDD: A decision diagram structure for reversible and quantum circuits. In *ISMVL*, pages 30–30, 2006.

[19] A. Mishchenko, S. Chatterjee, and R. K. Brayton. DAG-aware AIG rewriting a fresh look at combinational logic synthesis. In *DAC*, pages 532–535, 2006.

[20] A. Mishchenko and M. Perkowski. Fast heuristic minimization of exclusive-sums-of-products. Reed-Muller Workshop, 2001.

[21] M. A. Nielsen and I. L. Chuang. *Quantum Computation and Quantum Information*. Cambridge University Press, New York, NY, USA, 2000.

[22] A. P. E. Rosiello, F. Ferrandi, D. Pandini, and D. Sciuto. A hash-based approach for functional regularity extraction during logic synthesis. In *ISVLSI*, pages 92–97, 2007.

[23] M. Saeedi and I. L. Markov. Synthesis and optimization of reversible circuits: A survey. *ACM Comput. Surv.*, 45(2):21:1–21:34, 2013.

[24] Y. Sanaee and G. W. Dueck. Generating Toffoli networks from ESOP expressions. In *Communications, Computers and Signal Processing, 2009. PacRim 2009. IEEE Pacific Rim Conference on*, pages 715–719, 2009.

[25] P. W. Shor. Polynomial-time algorithms for prime factorization and discrete logarithms on a quantum computer. *SIAM J. Comp.*, 26(5):1484–1509, 1997.

[26] M. Soeken, S. Frehse, R. Wille, and R. Drechsler. RevKit: An open source toolkit for the design of reversible circuits. In *RC*, pages 64–76, 2012. RevKit is available at www.revkit.org.

[27] M. Soeken, L. Tague, G. W. Dueck, and R. Drechsler. Ancilla-free synthesis of large reversible functions using binary decision diagrams. *JSC*, 73:1–26, 2016.

[28] M. Soeken, R. Wille, C. Hilken, N. Przigoda, and R. Drechsler. Synthesis of reversible circuits with minimal lines for large functions. In *ASP-DAC*, pages 85–92, 2012.

[29] M. Soeken, R. Wille, O. Keszocze, D. M. Miller, and R. Drechsler. Embedding of large Boolean functions for reversible logic. *JETC*, 12(4), 2015.

[30] M. Veldhorst, C. H. Yang, J. C. C. Hwang, W. Huang, J. P. Dehollain, J. T. Muhonen, S. Simmons, A. Laucht, F. E. Hudson, K. M. Itoh, A. Morello, and A. S. Dzurak. A two-qubit logic gate in silicon. *Nature*, 526, 2015.

[31] G. F. Viamontes, M. Rajagopalan, I. L. Markov, and J. P. Hayes. Gate-level simulation of quantum circuits. In *Asia and South Pacific Design Automation Conference*, pages 295–301, 2003.

[32] R. Wille and R. Drechsler. BDD-based synthesis of reversible logic for large functions. In *DAC*, pages 270–275, 2009.

[33] R. Zhang, P. Gupta, L. Zhong, and N. K. Jha. Synthesis and optimization of threshold logic networks with application to nanotechnologies. In *DATE*, pages 904–909, 2004.