npj | Quantum Information

www.nature.com/npjqi

ARTICLE OPEN

Check for updates

Xor-And-Inverter Graphs for Quantum Compilation

Giulia Meuli[1,2]✉, Mathias Soeken[1] and Giovanni De Micheli[1]

Quantum compilation is the task of translating a high-level description of a quantum algorithm into a sequence of low-level quantum operations. We propose and motivate the use of Xor-And-Inverter Graphs (XAG) to specify Boolean functions for quantum compilation. We present three different XAG-based compilation algorithms to synthesize quantum circuits in the Clifford + T library, hence targeting fault-tolerant quantum computing. The algorithms are designed to minimize relevant cost functions, such as the number of qubits, the T-count, and the T-depth, while allowing the flexibility of exploring different solutions. We present novel resource estimation results for relevant cryptographic and arithmetic benchmarks. The achieved results show a significant reduction in both T-count and T-depth when compared with the state-of-the-art.

npj Quantum Information (2022)8:7 ; https://doi.org/10.1038/s41534-021-00514-y

INTRODUCTION

Different programming languages are currently available to program quantum computers at a high level of abstraction, with the purpose of enabling a wide community to exploit their exceptional computation capabilities. Relevant examples are: Q# (Microsoft)[1], Qiskit (IBM)[2], PyQuil/Quil (Rigetti)[3], Circ (Google)[4], Quipper[5], Scaffold/ScaffCC[6], and ProjectQ[7]. These languages require fast and reliable methods to compile the program into hardware-specific low-level quantum operations. The compilation result is evaluated by the number of qubits used, as well as by the number and the entity of low-level operations obtained.

Many quantum algorithms, such as Grover's[8], Shor's[9] and HHL[10], require the computation of some combinational logic functions, e.g., arithmetic functions, which usually need large amounts of resources to be computed. Methods capable of generating quantum circuits for such logic designs are needed to run these algorithms on a quantum computer. For example, HHL requires the reciprocal operation, which causes a significant overhead in the number of qubits with respect to the other components of the algorithm. In some cases, the resources required to perform logic operations may dominate the overall resources and exceed the available computing power. Besides, quantum circuits performing combinational logic, called *oracles*, find application in post-quantum cryptography. It has been shown how Grover's algorithm can be used to break symmetric encryption schemes such as the Advanced Encryption Standard (AES), if the quantum circuit for the encryption function is known[11,12]. The number of resources required to break a newly proposed post-quantum encryption scheme depends on the resources required to build the corresponding quantum oracle. Consider for example the categories for public-key schemes proposed by the National Institute of Standards and Technology (NIST) in their proposal to standardize post-quantum cryptography[13]. Shor's algorithm also requires combinational logic and can be used to construct quantum algorithms for integer factorization, finite field discrete logarithms, and elliptic curve discrete logarithms. As a consequence, cryptosystems based on these problems cannot be considered secure in a post-quantum environment.

Even if the technology is nowadays still far from achieving the system sizes and performances that these applications require, estimating the resources needed to perform combinational functions has a relevant impact on the design and applicability of advanced quantum algorithms. The resource footprint of these operations, e.g., a large number of quantum operations and qubits, can exceed the actual resources available, hence preventing some algorithms to be computed. Consequently, there is a large interest in compilation methods that minimize the impact of combinational logic on the cost of quantum algorithms.

Several research works focus on improving (often manually) quantum implementations of cryptographic functions. As Shor's algorithm can be used to break elliptic curve cryptography, authors of[14] have optimized the required quantum circuit that computes the costly elliptic curve scalar multiplication. The authors of ref. [11] present Clifford + T implementations of AES (key size 128, 192, and 256) used to evaluate the resources needed to run an exhaustive key search with Grover's algorithm. In ref. [15], authors present resource estimations of quantum pre-image attacks on SHA-2 and SHA-3. They present quantum oracles for SHA-256 and SHA3-256. They improve the reversible implementations derived in ref. [16] and evaluate the cost of running the attack on a surface code based fault-tolerant quantum computer. In ref. [17] authors focus on improving the implementation of the S-box of AES to simplify Grover based key search. Similarly, authors in ref. [18] provide implementations for SHA-256 and AES-128, result successively improved by Jaques et al. [12].

In this work, we focus on the problem of automatically compiling arbitrary logic functions for fault-tolerant quantum computing, starting from a multilevel logic network representation. With respect to the previously cited works, we do not rely on manual and design-specific optimizations. Our automatic compilation strategies are designed to minimize qubits and gates, with an emphasis on exploring the trade-off between the two cost functions. The algorithms are inspired by methods currently applied in classical multilevel logic synthesis—a 50 years old research field focused on optimization and mapping of combinational designs[19]. Algorithms and data structures developed in this field can be borrowed, adapted, and expanded to the synthesis of quantum circuits. In particular, we exploit a convenient graph-based data structure called Xor-And-Inverter Graphs (XAG). As we target fault-tolerant quantum computing, we compile into the

[1]Integrated Systems Laboratory, EPFL, Lausanne, Switzerland. [2]Synopsys Italia, Silicon Realization Group, Agrate Brianza, Italy. ✉email: meuli@synopsys.com

Clifford + T universal library and focus on the following cost functions: the T-count—the number of generated T gates; the T-depth—the maximum number of T gates to be performed sequentially, also referred to as number of T-stages; and the number of qubits. We identify how the characteristics of the network impact the resource footprint of the compiled circuit and elaborate on how the network could be modified to achieve better compilation results using state-of-the-art minimization strategies[20,21].

Logic networks are often used as convenient representation to develop scalable reversible synthesis algorithms[22–24]. A recent work[25] presents an automatic hierarchical synthesis method that leverages look-up table (LUT) decomposition. Such a method has the advantage of being applicable to any logic network, independently of the Boolean function implemented by its nodes. More importantly, it enables us to control the number of generated qubits: the network is decomposed into several single-output sub-networks whose results are stored into extra qubits. By controlling the size of the sub-networks, it is possible to control the extra qubits generated. Nevertheless, the method is not able to efficiently optimize the gate count. Typically, when the number of qubits is heavily constrained, the number of gates significantly increases. This happens because large sub-networks will be generated and, with no control on the Boolean functions they implement, they will likely be compiled into a large circuit. In addition, LUT decomposition causes a windowing effect: parts of the networks are prevented from being synthesized together, resulting in more gates. To address this issue, the work in ref. [26] implements an LUT decomposition strategy which allows some control on the grouped logic, reducing the T-count.

The present work is based on a different synthesis approach that enables better control over all the cost functions, which we introduced for the first time in ref. [27]. This approach is based on identifying repeated patterns in the network, which conveniently translate into quantum circuits with few gates. In particular, the graph is decomposed into parts that can be implemented by one single Toffoli gate. Hence, a direct correlation can be established between the features of the networks and the cost in terms of T gates (T-count and T-depth) and number of qubits.

In this work, we present all the latest improvements on XAG-based compilation, which reflect in the algorithms collected in the open-source library *caterpillar*. We propose XAG-based compilation as the method of choice to automatically synthesize quantum circuits implementing cryptographic and arithmetic logic functions with application in post-quantum cryptography and fault-tolerant quantum computing. Through the provided detailed description of the algorithms, the reader can identify (i) the most suited algorithm and (ii) the best XAG pre-processing steps to be used with respect to a specific compilation problem.

The first algorithm presented, which was originally proposed in ref. [27], minimizes the T-count by correlating it with the number of AND nodes in the XAG (multiplicative complexity). Indeed, the final circuit achieves the upper-bound in the number of T gates of four times the multiplicative complexity of the input network. We demonstrated in ref. [27] an average 20 × reduction in T-count with respect to LUT-based methods. The second algorithm proposed minimizes the T-depth by relating it to (i) the maximum number of levels in the graph with AND nodes, i.e., the multiplicative depth, and (ii) the number of AND nodes in the same level sharing input signals. This algorithm achieves a T-depth equal to the multiplicative depth of the graph and has been originally used in ref. [28] to synthesize designs with maximum 5 inputs. We provide a detailed algorithmic description of both algorithms. Furthermore, we present synthesis results for relevant cryptographic benchmarks (https://homes.esat.kuleuven.be/~nsmart/MPC/ and http://cs-www.cs.yale.edu/homes/peralta/CircuitStuff/CMT.html), which can serve as resource estimation for post-quantum attacks. Such results are compared with the state-of-the-art estimates available in the literature for some of the designs, showing improvement with respect to both T-count and T-depth. Differently from ref. [28], we provide resource estimation results for very large designs, proving the scalability of the proposed methods. We discuss and compare the results that the two methods achieve in addition to explaining how properties of the XAGs can be modified to tune the obtained results. For example, we identify the node scheduling as a key tool to minimize the number of qubits when using the second algorithm.

Finally, in this paper we propose a third compilation algorithm that performs quantum memory management to explore the trade-off between qubits and T-count. The number of available helper qubits can be selected as a parameter of the algorithm, which will return a valid compilation solution to not exceed the given qubit constraint, then an optimization procedure reduces the number of T gates. In particular, it exploits SAT solvers to find a strategy to fit the logic into a constrained number of qubits. The idea is to enable the reuse of helper qubits by uncomputing intermediate results, solving the so-called reversible pebbling game[29]. In a previous work[30] we introduced the problem of quantum memory management and proposed a solution based on SAT. With respect to the first attempt to apply this idea to XAGs in ref. [27], here we propose to work at a wider level of granularity. In other words, while the previous method was enabling computation and uncomputation of every single node in the XAG separately, in this approach we group selected sets of nodes together. This allows us to control the overhead in the number of gates generated when constraining the number of qubits. We present a SAT encoding that, by reducing the number of variables and the size of clauses, is applicable to larger designs and enables a second optimization algorithm to further improve the T-count of the compiled results. We demonstrate the ability of this method to trade-off qubits for gates on a selection of our benchmarks.

In classical logic synthesis, a good method is based on the synergy between data structure and algorithm, working together to minimize the target functions. Multilevel logic networks proved to be both scalable and compact data structures. For example, the And-Inverter Graph (AIG) is a popular network used both in academic and industrial frameworks[31,32].

In this work, we present different algorithms for the synthesis of quantum circuits that rely on the convenient representation of the logic as an XAG. This is a logic network over the gate basis $\{\wedge, \oplus, \neg\}$, meaning that each node of the network either computes the 2-input AND operation, the exclusive-OR operation, i.e., the 2-input XOR, or the inversion operation $\neg x = 1 \oplus x = \overline{x}$. We use $\overline{x}$ to denote the Boolean complement of $x = 1 - x$, and define $x^0 = \overline{x}$ and $x^1 = x$. A simple XAG computing the majority-of-three Boolean function is shown in Fig. 1a.

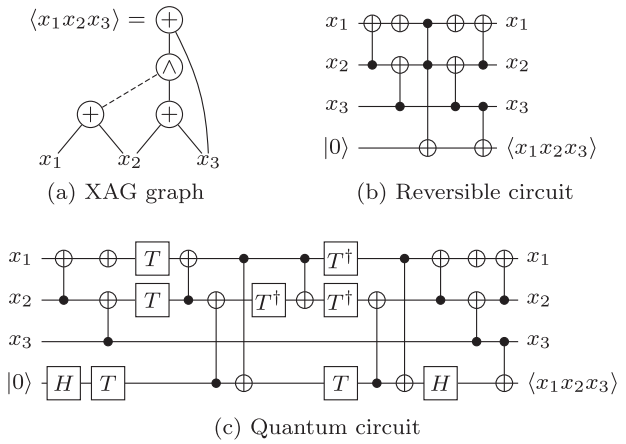
A Boolean chain is a formal notation for logic networks. Given primary inputs $x_1,\ldots x_n$, a logic network consisting of r local function is represented by a sequence called Boolean chain

$$x_i = f_i(x_{i_1}, \ldots, x_{i_{ar(f_i)}}) \quad \text{for} \quad n \le i \le n + r \tag{1}$$

where f_i is a gate function with $ar(f_i)$ inputs and $0 \le i_j < i$ for $1 \le j \le ar(f_i)$ are indexes to primary inputs or previous steps in the sequence, as defined in ref. [33]. An XAG logic network representing an n-variable Boolean function with inputs $x_1, \ldots, x_n$ is modeled as a Boolean chain with steps

$$x_i = x_{j(i)} \oplus x_{k(i)} \qquad \text{or} \qquad x_i = x_{j(i)}^{p(i)} \wedge x_{k(i)}^{q(i)}, \tag{2}$$

for $n < i \le n + r$, depending on whether the step computes the 2-input XOR or the 2-input AND operation, where r is the number of steps. The constant values $1 \le j(i) < k(i) < i$ point to input or



(a) XAG graph (b) Reversible circuit

(c) Quantum circuit

Fig. 1 The three steps perfomed to compile an XAG representing the majority-of-three function. a Specification; **b** corresponding reversible circuit; **c** corresponding quantum circuit.

previous steps in the chain. When a step computes the AND operation, the Boolean constants $p(i)$ and $q(i)$ are used to possibly complement the gate's fan-in. Please note that complemented inputs of XOR gates can be propagated to their outputs, hence we do not define $p(i)$ and $q(i)$ for the XOR steps. The value of a single-output function is computed by the last step of the chain $f = x_{n+r}^p$, which may be complemented. In the case of multi-output functions, there will be a set of steps that computes the function's values: $f_o = x_o^p$, where $o \in O$ is the list of all the output indices. We write $\circ_i = \wedge$, if step i computes an AND gate, and $\circ_i = \oplus$, if step i computes an XOR gate.

We define the *multiplicative complexity of the logic network* as the number of AND gates it contains: $\tilde{c} = |\{i|\circ_i = \wedge\}|$. We also define the *multiplicative complexity of the Boolean function*, which is the minimum number of AND nodes required to represent it as an XAG. Clearly, the multiplicative complexity of a network is an upper bound on the multiplicative complexity of the Boolean function it realizes.

In this work, we exploit the fact that every AND node acts on two multi-input parity functions. When the input to the AND node is either a primary input, another AND gate, or a network's output, the arity of this function is equal to 1. Formally, let the *linear transitive fan-in* of a node x_i in the logic network be defined using the recursive function

$$\mathrm{ltfi}(x_i) = \begin{cases} \{x_i\} & \text{if } i \le n \text{ or } \circ_i = \wedge \text{ or } i \in O, \\ \mathrm{ltfi}(x_{j(i)}) \triangle \mathrm{ltfi}(x_{k(i)}) & \text{otherwise}, \end{cases} \tag{3}$$

where 'Δ' denotes the symmetric difference of two sets. It is easy to see that all elements in ltfi(x_i) are either inputs, outputs, or steps that compute an AND gate. Figure 4 illustrate an AND node and its two linear transitive fan-in cones.

Example 1. The network in Fig. 1a, in which dotted lines represent inversion, implements the majority-of-three function $\langle x_1x_2x_3\rangle = x_1x_2 \vee x_1x_3 \vee x_2x_3$. The network corresponds to a Boolean chain with four steps:

$$x_4 = x_1 \oplus x_2, \qquad x_5 = x_2 \oplus x_3,$$
$$x_6 = \overline{x}_4 \wedge x_5, \qquad x_7 = x_3 \oplus x_6.$$

For this network

$$\mathrm{ltfi}(x_4) = \{x_1, x_2\},$$
$$\mathrm{ltfi}(x_5) = \{x_2, x_3\},$$
$$\mathrm{ltfi}(x_6) = \{x_6\},$$
$$\mathrm{ltfi}(x_7) = \{x_3, x_6\}.$$

Finally, we introduce the concept of *level* in the XAG network. Every step x_i of the network, with $1 \le i \le n + r$ is characterized by a quantity called *level* and defined as:

$$L(x_i) = \begin{cases} \max_{t \in C}(L(t)) + 1 \text{ with } C := \mathrm{ltfi}(x_j(i)) \cup \mathrm{ltfi}(x_k(i)), & \text{if } i > n \\ 0, & \text{otherwise} \end{cases} \tag{4}$$

In other words, a network's node x_i is at level $L(x_i) = l$ only if the node with the maximum level among all the ones in the linear transitive fan-in cones of x_i is at level $l - 1$. This means that only AND nodes and outputs count to define the depth of the network, because only AND and outputs nodes appear in the ltfi sets. We define $\max_{n<i\le n+r} L(x_i)$ as the *multiplicative depth* of the network.

In addition to providing a very compact representation for Boolean functions, XAG networks have another characteristic that makes them excellent data structures for quantum compilation: each node represents a logic function for which a convenient quantum circuit implementation exists. This allows us to recognize the existence of a dependency between the network characteristics, e.g., the multiplicative complexity/depth, and the synthesized quantum circuit. It is indeed possible to derive an upper bound on the number of expensive gates from characteristics of the XAG.

Given a logic network computing an n-variable Boolean function $f(x)$, a compilation algorithm finds a quantum circuit that implements the unitary operation

$$U_f : |x\rangle|y\rangle|0\rangle^k \mapsto |x\rangle|y \oplus f(x)\rangle|0\rangle^k, \tag{5}$$

where k is the number of extra qubits internally used by the circuit and restored back to $|0\rangle$, also referred to as *helper qubits*. This circuit is often called *oracle*. Automatic compilation of logic designs requires two steps, illustrated in Fig. 1: (i) transforming a possibly non-reversible Boolean function into a reversible quantum circuit, and (ii) translating the reversible circuit into a quantum circuit.

The first step is responsible of mapping the Boolean function into a reversible circuit. A reversible circuit is a logic representation characterized by a fixed number of lines that store inputs, outputs, and intermediate data, acted upon by reversible gates. For example, Fig. 1b shows the reversible circuit performing the function specified by the XAG in Fig. 1a. Such circuit is built using 2-input Toffoli gates, CNOT gates, and X gates (or NOT). The Toffoli gate is characterized by a set of two controls x_1, x_2 and by a single target y_1. It performs the transformation:

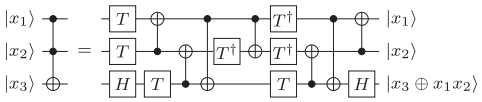
$$|x_1\rangle|x_2\rangle|y_1\rangle \mapsto |x_1\rangle|x_2\rangle|y_1 \oplus x_1x_2\rangle. \tag{6}$$

In other words, it inverts the target only if the logic AND of the two controls evaluates to one. In practice, if y_1 is initialized to $|0\rangle$, the Toffoli gate performs the AND operation. The CNOT is specified by a target and by a control qubit: it complements the target if the state of the control is $|1\rangle$. If applied on target in the state $|0\rangle$ the CNOT gate copies the state of the control.

Once the Boolean function is expressed using reversible gates, it needs to be compiled into a quantum circuit. Quantum circuits are a way to describe quantum programs: a sequence of operations performed on qubits, represented by quantum gates. We expect the reader to be familiar with the quantum circuit representation

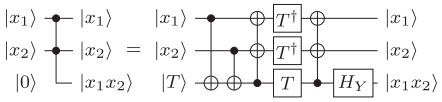
and gate abstractions and refer to ref. [34] for a detailed description. In fault-tolerant quantum computing, we consider gates from the Clifford + T universal library. This consists of the CNOT gate, the Hadamard gate (H), as well as the T gate, and its inverse $T^\dagger$. The T gate is particularly expensive to be applied. As a consequence, the T-count (number of T gates) is a good measure for the cost of a fault-tolerant implementation of a given quantum program[35,36].

Our algorithms exploit well known state-of-the-art quantum implementations of the 2-input Toffoli gate. The Toffoli gate has a Clifford + T implementation that requires 7 T gates[37], which is optimum[38,39]:



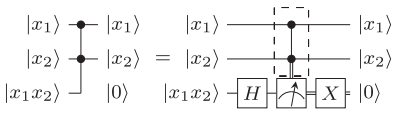
(7)

This implementation has been used to derive the quantum circuit for the majority-of-three function shown in Fig. 1c. When the Toffoli gate is computed on a qubit initialized to $|0\rangle$, it can be implemented using 4 T gates, with a T-depth of 2, and without requiring any additional qubit[40,41]:



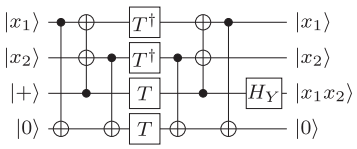
(8)

where $H_Y = SH$ and $|T\rangle = TH|0\rangle$. Besides, when the result of the Toffoli is uncomputed, this can be performed without the use of any T gate, exploiting measurement-based uncomputation[40], as shown:



(9)

There exists also another AND gate implementation with T-depth = 1, which combines the AND circuit from ref. [41] and the Toffoli gate implementation with T-depth = 1 in ref. [42]. The circuit requires one extra qubit with respect to the implementation in (8):



(10)

where $|+\rangle = H|0\rangle$.

RESULTS

In this section, we report the statistics of the quantum circuits generated by our XAG-based algorithms. We selected two publicly available benchmark suites, including arithmetic, cryptographic, e.g., AES, and floating point operation with applications in post-quantum cryptography and fault-tolerant quantum computing.

The first benchmark contains the best-known versions of logic networks in terms of multiplicative complexity and depth, collected by the Computer Security Resource Center (CSRC) at the National Institute of Standards and Technology (NIST). We synthesize: (i) finite field multiplication in $GF(2^6)$ using irreducible polynomial $x^6 + x^3 + 1$ (m × 6 × 31), multiplication in $GF(2^7)$ using irreducible polynomial $x^7 + x^4 + 1$ (m × 7 × 41) and using $x^7 + x^3 + 1$ (m × 7 × 31); (ii) binary multiplication with different input sizes n (bm_n); (iii) a 16-bit and a 8-bit S-box (s16, s8); (iv) finite field multiplication in $GF(2^8)$ using the AES polynomial $x^8 + x^4 + x^3 + x + 1$ (×8 × 4 × 31).

In addition, we evaluate our method on a set of circuits used in the context of Multi-Party Computation and Fully Homomorphic Encryption. From the benchmarks available online we synthesize: (i) block ciphers DES in its expanded and non-expanded variant (the latter meaning that the input key is assumed non-expanded); (ii) block cipher AES with 128, 192, and 256 key length; (iii) cryptographic hash functions MD5, Keccak, SHA-256, and SHA-512; (iv) arithmetic functions such as adders, multipliers, and comparators; (v) IEEE floating point operations. We pre-process the XAGs exploiting the toolbox to reduce the multiplicative complexity proposed by the authors of ref. [20]. This enables us to further improve the provided resource estimates for these designs.

Improving the T-count versus T-depth

Table 1 shows the synthesis results of the first two proposed algorithms. Alg. 1 minimizes the T-count, while Alg. 2 minimizes the T-depth without increasing the number of T gates, but relying on an increased number of additional qubits. The number of T gates achieved is equal to 4 times the multiplicative complexity of the network for both algorithms. The second algorithm obtains a T-depth equal to the multiplicative depth of the network. The last two columns of Table 1 compare the algorithms by reporting: the percentage of absolute change in T-depth (%Td) and in number of qubits (%Q) of Alg. 2 with respect to Alg. 1.

Figure 2 compares the results automatically obtained using Alg. 2 with some resource estimates available in the literature[11,12,15,17]. The comparison shows a significant reduction in both T-count and T-depth, while facing a less significant increase in number of qubits. Nevetheless, it is important to note that once mapped into an error-correcting code, T gates require a large amount of dedicated qubits. Note that the authors of ref. [17] only report the number of Toffoli gates and the Toffoli-depth. We obtain the corresponding T-count and T-depth by considering the Clifford+T implementation of the Toffoli gate with 7 T gates and a T-depth equal to 3, which is optimal[38].

Qubits/T-count trade-off

In this section, we show the results generated by our third algorithm to manage the memory resources during the compilation of the logic design. Our method allows us to force the compilation to synthesize a circuit with a limited number of helper qubits. Figure 3 shows the compilation results obtained setting the number of available helper qubits to different values, for a selection of designs. The plots show on the x-axis the number of qubits, and on the y-axis the obtained T-count. For every fixed number of qubits we report two points: the non-optimized and the optimized results. The latter obtained by running a post-optimization procedure encoded as a SAT problem on the initial (non-optimized) result. It can be seen how the procedure allows us to choose between different qubit/T-count trade-off solutions and how the optimization manages to minimize the T-count.

DISCUSSION

In the last section, we reported the specifics of quantum circuits compiled using our three XAG-based algorithms. In particular, the first two techniques achieve results that are predictable by inspecting the characteristics of the logic network. In details, given a logic network characterized by a multiplicative complexity $\tilde{c}$, i.e., the number of AND nodes, and by a multiplicative depth:

- both algorithms achieve a T-count equal to $4\tilde{c}$;
- Alg. 2 achieves a T-depth equal to the multiplicative depth;
- the qubit overhead to achieve such T-depth depends on the number of shared inputs in the linear transitive fan-ins of the AND nodes in a level.

This suggests that improving a network with respect to the named parameters can strongly and positively impact the

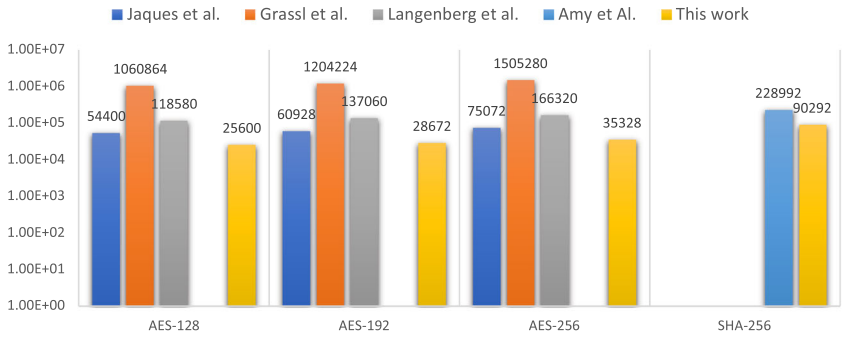
Table 1. Compilation results.

						Alg.1		Alg.2		Comparison	
benchmark	I	O	AND	XOR	Tc[a]	Td	Q	Td	Q	%Td	%Q
mcustom	16	8	27	79	108	8	51	1	116	12.50	227.45
mx6x31	12	6	27	30	108	6	45	1	112	16.67	248.89
mx7x41	14	7	40	44	160	7	61	1	164	14.29	268.85
mx7x31	14	7	40	45	160	7	61	1	164	14.29	268.85
s16	17	16	113	333	452	48	146	8	283	16.67	193.84
bm-10	20	19	52	102	208	12	89	1	218	8.33	244.94
bm-11	22	21	78	108	312	12	119	1	322	8.33	270.59
bm-12	24	23	81	126	324	12	126	1	332	8.33	263.49
bm-15	30	29	117	195	468	18	174	1	482	5.56	277.01
bm-20	40	39	208	314	832	24	285	1	850	4.17	298.25
bm-30	60	59	351	687	1404	36	468	1	1448	2.78	309.40
bm-40	80	79	624	1079	2496	48	781	1	2554	2.08	327.02
bm-50	100	99	676	1847	2704	72	873	1	2774	1.39	317.75
bm-60	120	119	1053	2253	4212	72	1290	1	4284	1.39	332.09
bm-70	140	139	1432	2985	5728	72	1709	1	5856	1.39	342.66
bm-80	160	159	1872	3494	7488	96	2189	1	7582	1.04	346.37
bm-90	180	179	1989	4561	7956	126	2346	1	8104	0.79	345.44
bm-100	200	199	2704	5143	10,816	144	3101	1	10,950	0.69	353.11
s8	9	8	32	81	128	28	49	6	63	21.43	128.57
x8x4x31	16	8	48	69	192	8	72	1	194	12.50	269.44
DES-expanded	832	64	9205	13,136	36,820	2070	10,101	214	10,352	10.34	102.48
DES-non-expanded	128	64	9048	13,092	36,192	2186	9240	202	9464	9.24	102.42
adder-32bit	64	33	32	150	128	33	129	32	130	96.97	100.78
adder-64bit	128	65	64	284	256	65	257	64	258	98.46	100.39
comparator-32bit-lt	64	1	92	95	368	21	156	20	182	95.24	116.67
comparator-32bit-lteq	64	1	92	97	368	20	156	19	182	95.00	116.67
md5	512	128	9367	29,729	37,468	7561	10,007	1283	10,619	16.97	106.12
mult-32x32	64	64	1689	4723	6756	324	1816	64	1816	19.75	100.00
Keccak-f	1600	1600	38,400	115,200	153,600	30,209	41,600	24	44,798	0.08	107.69
AES-128	256	128	6400	28,176	25,600	874	6976	60	7133	6.86	102.25
AES-192	320	128	7168	32,080	28,672	830	7808	72	7870	8.67	100.79
AES-256	384	128	8832	39,008	35,328	900	9536	84	9598	9.33	100.65
SHA-256	768	256	22,573	109,746	90,292	14,411	23,597	1607	23,597	11.15	100.00
SHA-512	1536	512	57,947	284,286	231,788	40,172	59,995	3304	59,995	8.22	100.00
FP-add	128	64	5346	5629	21,384	2460	5538	235	5541	9.55	100.05
FP-div	128	64	70,599	44,959	282,396	25,649	70,792	3604	72,563	14.05	102.50
FP-eq	128	64	315	356	1260	10	506	9	526	90.00	103.95
FP-f2i	64	64	1458	1421	5832	593	1586	94	1683	15.85	106.12
FP-mul	128	64	18,874	10,290	75,496	1988	19,066	118	20,907	5.94	109.66
FP-sqrt	64	64	76,925	57,165	307,700	44,782	77,054	6498	79,589	14.51	103.29

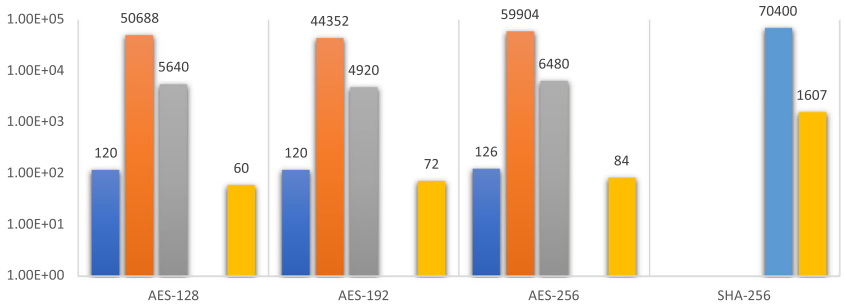
[a]Both algorithms achieve the same *T*-count.

synthesized quantum circuits, e.g., as done in ref. [21], to reduce the *T*-depth by reducing the multiplicative depth of the network.

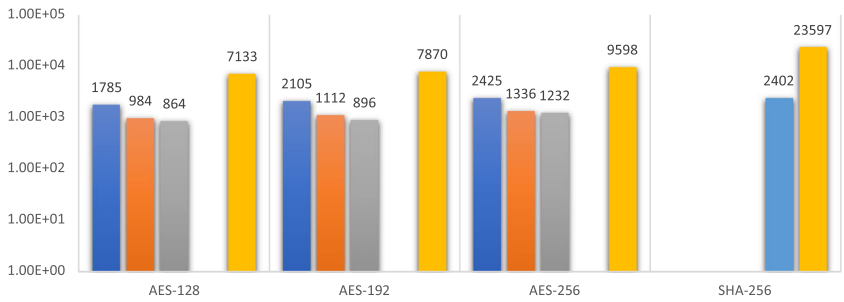
Inspecting the results of the comparison in Table 1 reveals a trade-off between *T*-depth and number of qubits. Indeed, while Alg. 1 is far from achieving the *T*-depth performances of Alg. 2, it requires fewer qubits. There are two reasons for the increase in qubits which characterizes Alg. 2. The first one is that it employs the AND implementation characterized by a single *T*-stage and presented in Section "Introduction" (10), which requires one qubit more than implementation (8) used by Alg. 1. This means that the compilation will request this extra qubit whenever a AND node is computed. In addition, the implementation of AND nodes used by the second algorithm is characterized by a *T* gate applied to the controls, as well as to the target qubit. For this reason, if two AND nodes share the same input signal, the corresponding quantum circuit will have a *T*-depth equal to 2, as each AND implementation will add a *T* gate to the shared qubit. If all the AND nodes at the same level of an XAG do not share any input, they can be computed within a single *T*-stage. In order to achieve this result, our second algorithm copies inputs that are shared among more



(a) T-count



(b) T-depth



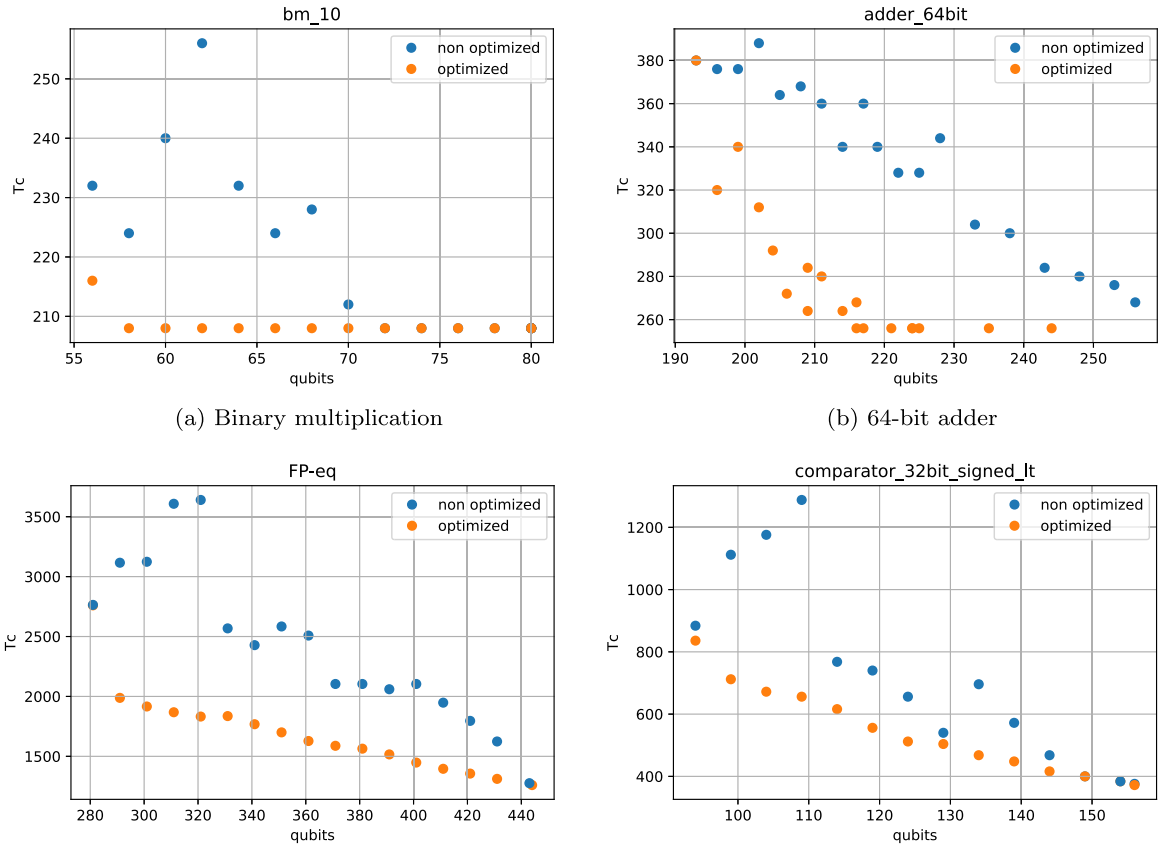
(c) Qubits

Fig. 2 Resource estimates for AES-128/192/256 and SHA-256 compared with the state-of-the art: Jaques et al.[12], Grassl et al.[11], Langenberg et al.[17] and Amy et al.[15]. a Histogram comparing the number of T gates; **b** histogram comparing the T-depth; **c** histogram comparing the number of qubits.

AND nodes in a level on new qubits. Hence, the compilation will request a new qubit whenever inputs are shared among AND nodes at the same level in the XAG. In conclusion, if we sum the number of AND nodes in a level with the number of shared inputs among them, we obtain a quantity equal to the number of helper qubits required to compile that level. Since helper qubits are cleaned-up after all the nodes in the level are computed, the level for which this amount is greater will dominate and give the total number of helper qubits for the synthesis of the entire network. Further details on the algorithm, including detailed pseudo-code, can be found in Section "Methods".

We chose to report in Table 1 the two extremes that can be reached using our constructive algorithms. It is also possible to obtain results 'in-between', i.e., a smaller improvement in T-depth and a smaller qubit overhead with respect to Alg. 2, e.g., by modifying Alg. 1 to use the implementation with T-depth equal to one. In addition, as the connectivity of each AND node in a level has an impact on the T depth, different results can be found by changing how the level of each node is computed. For example, it is possible to change the scheduling of the nodes to reduce the T depth while minimizing the qubit overhead of Alg. 2.

Our third algorithm focuses on exploring the trade-off between T-count and number of qubits. Figure 3 shows how our method is capable of providing different compiled solutions, by taking the number of helper qubits as a parameter. Our method finds the best way of reusing memory space, by computing and uncomputing helper qubits that store intermediate results. This problem corresponds to the reversible pebbling game. The problem complexity has been studied in ref. [43], where the author proves that finding the minimum number of pebbles is PSPACE-complete, as in the case of the non-reversible pebbling game. Besides, the problem is PSPACE-hard to approximate up to an additive constant[44]. An explicit asymptotic expression for the best time-space product is given in ref. [45]. This is a global problem, hard to approximate and decompose, hence difficult to be tackled by heuristic techniques. Here, the problem is encoded as a SAT problem and solved globally, returning a valid memory clean-up



(a) Binary multiplication

(b) 64-bit adder

(c) IEEE floating point equality

(d) 32-bit comparator

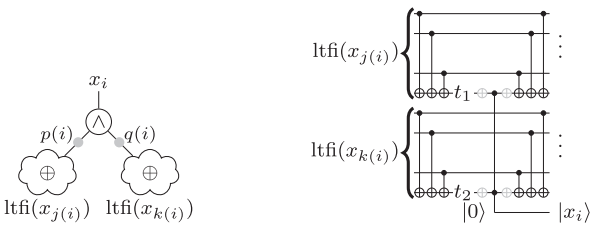
Fig. 3 Results of pebbling selected logic networks using different number of pebbles: comparison between optimized and non-optimized solutions. Results of pebbling a circuit implementing **a** binary multiplication; **b** a 64-bit addition; **c** IEEE floating point equality; **d** a 32-bit comparator.

strategy that guarantees the upper bound on the number of helper qubits while also aiming to minimize the *T*-count.

With respect to the SAT-based technique in ref. [27], the algorithm proposed in this work exploits a completely different SAT encoding, which is more compact in both number of variables and clauses. With this method it is possible to obtain competitive results for larger designs while guaranteeing better results for smaller designs. For example, consider the compilation of the small design s8 on 20 helper qubits: our method achieves a *T*-count of 164 while the results in ref. [27] show a *T*-count of about 280.

In Fig. 3 we show non-optimized versus optimized pebbling solutions. The non-optimized solution is provided by the SAT solver without any constraints on the number of *T* gates generated. The optimized solution is obtained starting from the initial solution and running optimization rounds, which iteratively add clauses to the SAT problem to minimize the *T*-count. The more time is spent in the optimization procedure the better the solution. The optimized points shown in Fig. 3 are either optimal or the best result found after 1 and a half hours of running the optimization procedure on a machine with two Intel Xeon E5-2680 v3 (Haswell) CPUs with 2.5 GHz clock frequency and 16 GB of main memory.

The optimization procedure removes unnecessary steps that the solver may insert in the solution. Indeed, none of the clauses used to encode the problem prevents the solver to uncompute nodes even if the limit in pebbles is not reached. Preventing this at the encoding level requires a non-practical increase in the size of the SAT problem. The optimization reveals the trade off between qubits and *T*-count.



(a) AND step in logic network

(b) Quantum circuit implementing an AND step

Fig. 4 Illustration of the general idea in which the fan-in nodes of an AND gate are considered as large XOR gates, computed in-place using CNOT gates. a An AND step in an XAG network; **b** corresponding compiled quantum circuit.

METHODS

Algorithm 1: minimizing the T-count

Our first algorithm achieves an upper bound on the number of *T* gates that is proportional to the multiplicative complexity of the input network $\tilde{c}$. Indeed, the final quantum circuit has $4\tilde{c}T$ gates.

The key insight is that each AND node in the logic network is driven by two multi-input parity functions of variables which are either inputs or other AND nodes in the lower levels of the logic network. Figure 4 shows the node x_i and the two parity functions with the respective linear

transitive fan-ins. The polarity variables $p(i)$ and $q(i)$ take into account possible inversion of the inputs of the AND node. The pseudo-code of the algorithm is provided by Alg. 1. Since the algorithm dedicates one helper qubit for each node of the XAG to store its computed Boolean function, we use nodes' identifiers, e.g. x_i, as parameters for quantum operations, e.g., NOT(x_i), meaning that the operation is performed on the corresponding qubits.

Lines 19–22 show that, at first, it computes all the steps of the network that perform the AND (or compute an output) using the function *compute*. Then all the intermediate results are restored to $|0\rangle$ by uncomputing '*compute*'. In lines 23–24 NOT gates are placed on negated outputs. The function *compute* (lines 2–18) builds the circuit for each step x_i as illustrated in Fig. 4. In particular, it identifies two qubits corresponding to nodes in the ltfi cones that are not shared between the cones, namely t_1 and t_2. Then, the parity functions are computed in-place onto these qubits t_1 and t_2. Then, the complemented edges are evaluated and NOT gates are applied if necessary (see Fig. 4). In lines 13–14 the step x_i is finally computed on a new qubit, using a CNOT gate in case of an XOR output or the implementation of the AND node described in (8), which has *T*-count equal to 4 and *T*-depth equal to 2, otherwise. Finally, the parity functions are uncomputed.

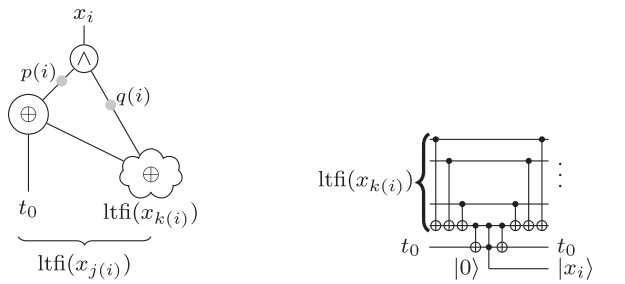
Algorithm 1. Low T-count compilation algorithm.

Input: Logic network with gates $x_{n+1}, \ldots, x_{n+r}$
Output: Quantum circuit for U_f

1 **function** *compute*(x_i) **is**
2 set $j \leftarrow j(i)$, $k \leftarrow k(i)$;
3 set $L_1 \leftarrow \mathrm{ltfi}(x_j)$, $L_2 \leftarrow \mathrm{ltfi}(x_k)$;
4 **if** $L_1 \subseteq L_2$ **then**
5 swap $L_1 \leftrightarrow L_2$ and $p \leftrightarrow q$;
6 request_helper(x_i);
7 let t_1 in $L_1 \setminus L_2$;
8 let t_2 in L_2;
9 $\mathrm{CNOT}(x, t_1)$ for all $x \in L_1 \setminus \{t_1\}$;
10 $\mathrm{CNOT}(x, t_2)$ for all $x \in L_2 \setminus \{t_2\}$;
11 **if** $p(i)$ **then** $\mathrm{NOT}(t_1)$;
12 **if** $q(i)$ **then** $\mathrm{NOT}(t_2)$;
13 **if** $o_i = \wedge$ **then** $\mathrm{AND}_{\text{T-depth}=2}(t_1, t_2, x_i)$;
14 **else** $\mathrm{XOR}(t_1, t_2, x_i)$;
15 **if** $p(i)$ **then** $\mathrm{NOT}(t_2)$;
16 **if** $q(i)$ **then** $\mathrm{NOT}(t_1)$;
17 $\mathrm{CNOT}(x, t_2)$ for all $x \in L_2 \setminus \{t_2\}$;
18 $\mathrm{CNOT}(x, t_1)$ for all $x \in L_1 \setminus \{t_1\}$;
19 **for** $i = n+1, \ldots, n+r$ *where* $\circ_i = \wedge$ *or* $i \in O$ **do**
20 compute(x_i);
21 **for** $i = n+r, \ldots, n+1$ *where* $\circ_i = \wedge$ *and* $i \notin O$ **do**
22 compute†(x_i);
23 **for** x_o^p *with* $o \in O$ **do**
24 **if** p **then** $\mathrm{NOT}(x_o)$;

Note that we assume that $L_1 \neq L_2$. If this is not the case, it means that the functions computed by fan-in to the AND gate are equal, making the AND gate redundant. Also, note that the intersection of L_1 and L_2 may not be empty. Since we want to compute the value of L_1 in-place on some signal $t_1 \in L_1$, we must ensure that $L_1 \nsubseteq L_2$. If the latter condition applies, it is sufficient to swap L_1 and L_2.

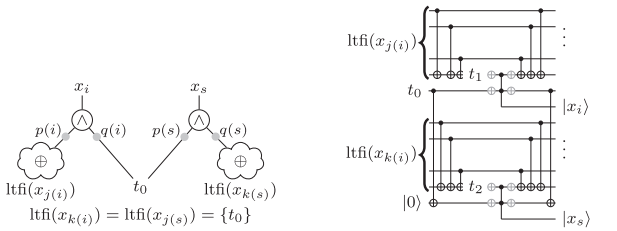
In addition, when $L_2 \subseteq L_1$, the value computed by L_2 could be reused to compute L_1. This is achieved by modifying the elements in L_1 such that $L_1 = (L_1 \backslash L_2) \cup \{x_k\}$. An example is shown in Fig. 5. In this case ltfi(x_j) includes ltfi(x_k) and ltfi(x_j)\ltfi(x_k) = $\{t_0\}$. This leads to a reduction in the number of CNOT operations.



(a) AND step with shared logic

(b) Quantum circuit implementing the step

Fig. 5 A special configuration with one transitive fan-in included in the other. a A special AND step in an XAG network; **b** corresponding compiled quantum circuit.



(a) XAG level with two AND nodes and a shared input

(b) Quantum circuit construction for the level

Fig. 6 Compilation of an XAG level with two AND nodes using algorithm 2. a XAG level with AND nodes x_i and x_s; **b** compiled quantum circuit with a single *T*-stage.

Algorithm 2: minimizing the T-depth

Our second algorithm targets the reduction of the *T*-depth. Unlike the previous algorithm, it uses the implementation of the AND operation that has 4 T gates, 4 qubits, and 1 *T*-stage (10).

We refer to $X_l = \{x_i | L(x_i) = l\}$, as the set of all the nodes at level *l*. The key idea is that if two AND nodes in the same level do not share any of their input in the ltfi sets, then they can be computed with only one *T*-stage using implementation (10). Obviously, this is not always the case, as AND nodes often share the same inputs. To overcome this problem, the algorithm copies every overlapping set of inputs on a new helper qubit. This procedure, described in Alg. 2, obtains circuits with a number of *T*-stages equal to the multiplicative depth of the networks. While the previously described algorithm proceeds in topological order, this one proceeds level by level (see lines 10–17). For each level, the function *copy_overlaps* assigns to each node a set of two qubits on which it computes the parities of the two fan-in cones, defining the mapping *CP*. If the node shares some inputs with another, a new qubit will be assigned to compute the corresponding parity function, otherwise a qubit corresponding to a node in the fan-in cone is used. This means that if a node $x_i \in X_l$ has inputs t_1, t_3, t_5 (on qubits q_1, q_3, q_5) in common with node $x_j \in X_l$, then a new qubit q_i will be used as target of three CNOT gates with the shared input qubits as controls. As it can be seen in line 11, the copies are performed before computing any of the nodes in the level, thus allowing the actual AND implementations to act on non-overlapping qubits, resulting in a single *T*-stage. Once the copies are being computed, each node is passed to the function *compute_on_copies* (lines 1–9) which uses the qubits associated by the mapping *CP* to each fan-in parity function as controls to compute the AND. Once all AND nodes in the level are computed, the parities are uncomputed (lines 14). Finally the levels in the XAG are uncomputed from top to bottom. Every node, independently from having shared fan-ins can be uncomputed without using copies (lines 15–17), applying the function *compute* defined in Alg. 1. Finally in lines 18–end NOT gates are placed on complemented outputs. An illustrative example is shown in Fig. 6, where the algorithm is applied to a simple level $X_l = x_i, x_s$ with one overlapping input t_0, such that $\mathrm{ltfi}(x_{j(i)}) \cap \mathrm{ltfi}(x_{k(s)}) = \{\}$ and $\mathrm{ltfi}(x_{j(s)}) = \mathrm{ltfi}(x_{k(i)}) = \{t_0\}$. The figure shows how the overlapping input is copied to a new qubit before computing the parity functions: then the two AND can be computed in parallel with a *T*-depth equal to 1.

Algorithm 2. Low T-depth compilation algorithm.

Input: Logic network with gates $x_{n+1}, \ldots, x_{n+r}$
Output: Quantum circuit for U_f
function *compute_on_copies*(x_i, *CP*) **is**
  set $t_1, t_2 \leftarrow \mathrm{CP}[i]$;
  request_helper(x_i);
  if $p(i)$ **then** NOT(t_1);
  if $q(i)$ **then** NOT(t_2);
  if $o_i = \wedge$ **then** $\mathrm{AND}_{\text{T-depth}=1}(t_1, t_2, x_i)$;
  else XOR(t_1, t_2, x_i);
  if $p(i)$ **then** NOT(t_2);
  if $q(i)$ **then** NOT(t_1);
for $l = 1 \ldots \tilde{d}$ **do**
  CP $\leftarrow$ copy_overlaps(X_l);
  for $x_i \in X_l$ **do**
    compute_on_copies(x_i, CP);
  copy_overlaps†(X_l);
for $l = \tilde{d}, \ldots, 1$ **do**
  for $x_i \in X_l$ *where* $i \notin O$ **do**
    compute†(x_i);
for x_o^p *with* $o \in O$ **do**
  if p **then** NOT(x_o);

Algorithm 3: minimizing the number of qubits

All the algorithms described so far compute and uncompute every AND node at most once, and the compiled circuit is uniquely determined by the features of the input network. In this section, we show a method that, instead, allows us to explore the solution space, by enabling to compute and uncompute nodes several times.

The third algorithm seeks the best strategy to uncompute the intermediate results in order to optimize the memory usage. The problem is equivalent to the reversible pebbling game. The game is played on a directed acyclic graph (DAG) using a limited number of pebbles. The player places or removes pebbles from the DAG nodes according to certain rules: a pebble can be placed (removed) from a node only if all the inputs of that node have a pebble. The game is won when pebbles are only placed on the network's output. The set of moves that leads to a winning configuration is called *pebbling strategy*. Every pebble in the game corresponds to a helper qubit. The move of placing a pebble on a node corresponds to computing the logic of that node on this helper qubit. When a pebble is removed, it corresponds to uncomputing the value stored on the helper qubit. As a consequence, the pebbling strategy directly corresponds to a set of compute/uncompute operations. The definition of a winning configuration (no pebbles on internal nodes) guarantees that performing this set of operations uncomputes all intermediate results. As demonstrated in ref. [30], SAT solvers can be used to solve the reversible pebbling game and find a synthesis strategy for any Boolean function represented using a DAG.

The compilation problem is transformed into the following problem:

Problem 1

Given a DAG and a number of pebbles, find a valid pebbling strategy using the minimum number of moves.

To address this problem using a SAT solver, it needs to be decomposed into many SAT problems:

Problem 2

Given a DAG and P pebbles, does a valid pebbling strategy with K moves exist?

The solver can either find a solution and return a pebbling strategy, or state that no solution exists. In order to solve problem 1, when the SAT solver returns *unsat*, K is incremented and the solver is asked to find a strategy again. This is done until a satisfying solution is found. Since K is incremented at each step, once a solution is found, it is guaranteed to be the one with the smallest K.

SAT encoding. Here we give a quick overview of the basic encoding. The input DAG $G = (V, E)$ figures nodes computing output values and we refer to them as elements of the set $O \subseteq V$. Note that the primary inputs are not nodes of the DAG. Problem 2 is encoded in terms of the pebble state variables $p_{v,i}$. For $v \in V$ and $0 \le i \le K$, those are Boolean variables that evaluate to true if the node v is pebbled at time i. Note that the SAT formula encodes $K+1$ pebble configurations with K steps describing the transition from one configuration to the other. The following set of clauses describes the reversible pebbling problem:

- *Initial and final clauses*. At time 0 all the nodes are unpebbled and at time K all the outputs need to be pebbled and all the intermediate results unpebbled

$$\bigwedge_{v \in V} \overline{p}_{v,0} \wedge \bigwedge_{v \in O} p_{v,K} \wedge \bigwedge_{v \notin O} \overline{p}_{v,K}$$

- *Move clauses*. If a node is pebbled or unpebbled at time $i+1$, then all its children are pebbled at time i and time $i+1$:

$$\bigwedge_{i=1}^{K} \bigwedge_{(v,w) \in E} ((p_{v,i} \oplus p_{v,i+1}) \rightarrow (p_{w,i} \wedge p_{w,i+1}))$$

- *Cardinality clauses*. At each step, at most P pebbles are used:

$$\bigwedge_{i=0}^{K} \left(\sum_{v \in V} p_{v,i} \le P\right)$$

Example 2. Figure 7 illustrates how a network with only AND nodes can be compiled as a reversible network of Toffoli gates out of a pebbling solution with 3 pebbles and 6 steps. Note that the final circuit will use only 2 helper qubits, which is the number of pebbles used, minus the number of outputs. The overall width will be equal to 7: the number of inputs plus the number of pebbles.

XAGs are DAGs in which each node computes the AND or the XOR function. It follows that it is possible to play the reversible pebbling game directly on the XAG, as done in ref. [27]. Nevertheless, this does not exploit the structural properties of the XAG. In addition, the SAT encoding required for a similar approach must be capable of discriminating between the different properties of the XAG node. For example, several clauses are required to enable in-place computing of XOR nodes. The resulting SAT problem features many variables and clauses and is only applicable to small designs.

For these reasons, we choose to construct a different DAG from the XAG, which we call *abstract graph*. Each AND node (and its two input parity functions) corresponds to a box node of the abstract graph, as shown in and Fig. 8. Once a strategy for pebbling the abstract graph is found, each time a pebble is placed on a box node which compresses x_i the *compute*(x_i) function will be called, while whenever a pebble is removed from a node, the *compute*†(x_i) function will be called to uncompute the node.

Optimizing the pebbling solution

While the XAG is compressed into the abstract graph we lose some information about the number of quantum gates required to compute each node. Indeed, the strategy found would not take into account the fact that one box node requires more gates to be performed than another. In addition, the SAT encoding of the standard reversible pebbling game does not include any clause that controls the number of moves, which reflects in the number of generated T gates. An optimization step is introduced to overcome both problems.

The key idea is that it is possible to associate a weight with each box node of the abstract graph w_v, which is equal to the number of inputs to the node itself. Indeed, the number of inputs are related to the number of CNOT gates that are needed to compute the parity functions 'hidden' in the compressed node. Then, we define a new set of variables for the SAT encoding: activation variables $a_{v,i}$. For $v \in V$ and $0 < i \le K$, those are Boolean variables that evaluate to true if the node v has changed its state at time i. Once a weight-agnostic solution has been found, the following quantity

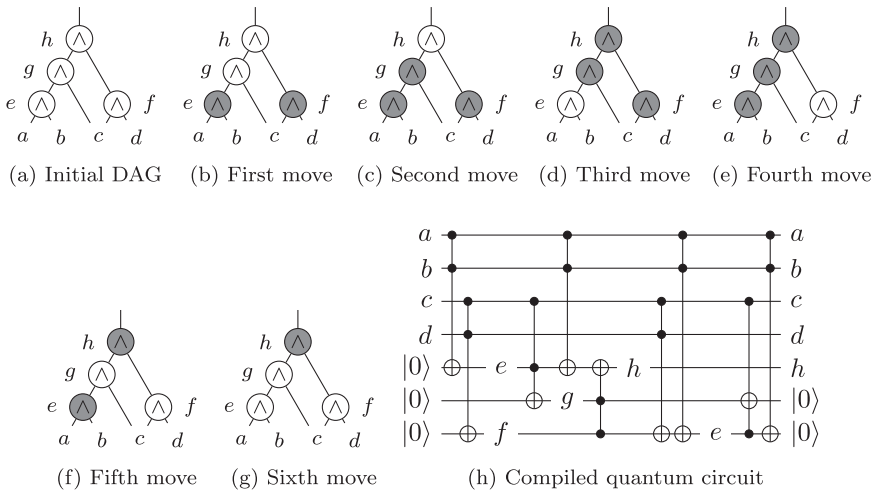


Fig. 7 Illustration of a pebbling strategy using 3 pebbles and 6 moves. a Input DAG; **b**–**g** pebbling moves where dark nodes are pebbled; **h** the corresponding compiled reversible circuit of Toffoli gates.

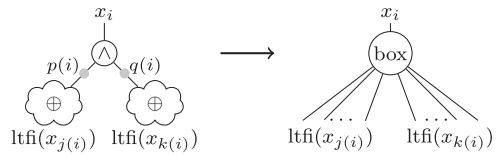


Fig. 8 Illustration of how sections of the XAG are compressed in a box node of the abstract network.

represent the total weight of the strategy:

$$W_s = \sum_{i=1}^{K} \sum_{v \in V} w_v a_{v,i} \tag{11}$$

The SAT solver is then asked to find a solution with a total weight $W = W_s - 1$ by adding a cardinality clause that expresses equation (11). This procedure is repeated until the solver returns 'unsat' or hits a timeout.

As shown in the result section, this optimization procedure succeeds at reducing the number of T gates with respect to the initial solution. This result can be achieved even if every node has weight equal to one. Indeed, the optimization introduces a cardinality constraint on the activation variables, hence eliminates all the pebbling moves that are not fundamental to terminate the game. As a consequence, fewer helper qubits are required. If the weights are set to reflect the actual size of the parity functions, then the number of CNOT in the solution is reduced.

DATA AVAILABILITY

The circuits we synthesized have been collected by the NIST and the University of Yale (http://cs-www.cs.yale.edu/homes/peralta/CircuitStuff/CMT.html) and by the Department of Electrical Engineering (ESAT) at KU Leuven (https://homes.esat.kuleuven.be/~nsmart/MPC/). For some entries of our benchmark we used circuit implementations with low multiplicative complexity obtained at EPFL and available online at https://github.com/lsils/date2020_experiments.

CODE AVAILABILITY

All the algorithms that we discussed in this work are part of the C++ open-source library *caterpillar* (https://github.com/gmeuli/caterpillar), which is one of the LSI logic synthesis libraries[46].

Received: 14 August 2020; Accepted: 3 December 2021;
Published online: 27 January 2022

REFERENCES

1. Svore, K. M. et al. Q#: Enabling scalable quantum computing and development with a high-level DSL. In *Real World Domain Specific Languages Workshop*, 7:1–7:10 (2018).
2. Aleksandrowicz, G. et al. Qiskit: An Open-source Framework for Quantum Computing (2019). Zenodo. https://doi.org/10.5281/zenodo.2562111.
3. Smith, R. S., Curtis, M. J. & Zeng, W. J. A practical quantum instruction set architecture. Preprint at https://arxiv.org/abs/1608.03355 (2017).
4. Ho, A. & Bacon, D. Announcing Cirq: An open source framework for NISQ algorithms. *Google AI Blog* (2018).
5. Green, A. S., Lumsdaine, P. L., Ross, N. J., Selinger, P. & Valiron, B. Quipper: a scalable quantum programming language. In *ACM SIGPLAN Conference on Programming Language Design and Implementation*, 333–342 (2013).
6. Javadi-Abhari, A. et al. ScaffCC: a framework for compilation and analysis of quantum computing programs. *Proceedings of the 11th ACM Conference on Computing Frontiers, CF 2014* (2014).
7. Steiger, D. S., Häner, T. & Troyer, M. ProjectQ: an open source software framework for quantum computing. *Quantum* **2**, 49 (2018).
8. Grover, L. K. Quantum computers can search arbitrarily large databases by a single query. *Phys. Rev. Lett.* **79**, 4709 (1997).
9. Shor, P. W. Polynomial-time algorithms for prime factorization and discrete logarithms on a quantum computer. *SIAM Rev.* **41**, 303–332 (1999).
10. Harrow, A. W., Hassidim, A. & Lloyd, S. Quantum algorithm for linear systems of equations. *Phys. Rev. Lett.* **103**, 150502 (2009).
11. Grassl, M., Langenberg, B., Roetteler, M. & Steinwandt, R. Applying Grover's algorithm to AES: quantum resource estimates. In: *Post-Quantum Cryptography*. PQCrypto 2016 (ed. Takagi, T.), vol. 9606, 29–43 (2016).
12. Jaques, S., Naehrig, M., Roetteler, M. & Virdia, F. Implementing grover oracles for quantum key search on AES and LowMC. In *Annual Int'l Conf. on the Theory and Applications of Cryptographic Techniques*, 280–310 (Springer, 2020).
13. NIST. Submission requirements and evaluation criteria for the post-quantum cryptography standardization process (2016). Online at https://csrc.nist.gov/CSRC/media/Projects/Lightweight-Cryptography/documents/final-lwc-submission-requirements-august2018.pdf.
14. Häner, T., Jaques, S., Naehrig, M., Roetteler, M. & Soeken, M. Improved quantum circuits for elliptic curve discrete logarithms. In *Int'l Conf. on Post-Quantum Cryptography*, 425–444 (Springer, 2020).
15. Amy, M. et al. Estimating the cost of generic quantum pre-image attacks on sha-2 and sha-3. In: *Selected Areas in Cryptography – SAC 2016*. (eds. Avanzi, R. & Heys, H.), vol. 10532, 317–337 (2017).
16. Parent, A., Roetteler, M. & Svore, K. M. Reversible circuit compilation with space constraints. Preprint at https://arxiv.org/abs/1510.00377 (2015).
17. Langenberg, B., Pham, H. & Steinwandt, R. Reducing the cost of implementing the advanced encryption standard as a quantum circuit. *IEEE Trans. Quantum Eng.* **1**, 1–12 (2020).
18. Kim, P., Han, D. & Jeong, K. C. Time-space complexity of quantum search algorithms in symmetric cryptanalysis: applying to AES and SHA-2. *Quantum Inf. Process.* **17**, 339 (2018).
19. Brayton, R. K., Hachtel, G. D. & Sangiovanni-Vincentelli, A. L. Multilevel logic synthesis. *Proc. IEEE* **78**, 264–300 (1990).

20. Testa, E., Soeken, M., Riener, H., Amaru, L. & De Micheli, G. A logic synthesis toolbox for reducing the multiplicative complexity in logic networks. In *Design, Automation and Test in Europe Conference* (2020).
21. Häner, T. & Soeken, M. Lowering the T-depth of quantum circuits by reducing the multiplicative depth of logic networks. Preprint at https://arxiv.org/abs/2006.03845 (2020).
22. Rawski, M. Application of functional decomposition in synthesis of reversible circuits. In *Reversible Computation. RC 2015*. (eds. Krivine, J. & Stefani, J. B.), vol. 9138, 285–290 (2015).
23. Markov, I. L. & Saeedi, M. Faster quantum number factoring via circuit synthesis. *Phys. Rev. A* **87**, 012310 (2013).
24. Shende, V. V., Prasad, A. K., Markov, I. L. & Hayes, J. P. Synthesis of reversible logic circuits. *IEEE Trans. Comput. Aided Design Integrated Circuits Syst.* **22**, 710–722 (2003).
25. Soeken, M., Roetteler, M., Wiebe, N. & De Micheli, G. LUT-based hierarchical reversible logic synthesis. *IEEE Trans. Comput. Aided Design Integrated Circuits Syst.* **38**, 1675–1688 (2018).
26. Meuli, G., Soeken, M., Roetteler, M. & De Micheli, G. ROS: Resource constrained oracle synthesis for quantum circuits. In *Quantum Physics and Logic* (2019).
27. Meuli, G., Soeken, M., Campbell, E., Roetteler, M. & De Micheli, G. The role of multiplicative complexity in compiling low T-count oracle circuits. *Int'l Conf. on Computer-Aided Design* (2019).
28. Meuli, G., Soeken, M., Roetteler, M. & De Micheli, G. Enumerating optimal quantum circuits using spectral classification. In *Int'l Symp. on Circuits and Systems* (2020).
29. Bennett, C. H. Time/space trade-offs for reversible computation. *SIAM J. Comput.* **18**, 766–776 (1989).
30. Meuli, G., Soeken, M., Roetteler, M., Bjorner, N. & Micheli, G. D. Reversible pebbling game for quantum memory management. In *Design, Automation and Test in Europe Conference*, 288–291 (2019).
31. Brayton, R. & Mishchenko, A. ABC: An academic industrial-strength verification tool. In *Int'l Conf. on Computer Aided Verification*, 24–40 (Springer, 2010).
32. Synopsys. Design compiler graphical. Online at https://www.synopsys.com/implementation-and-signoff/rtl-synthesis-test/design-compiler-graphical.html (2020). Accessed Apr 2020.
33. Knuth, D. E. *The Art of Computer Programming,* vol. 4A (Addison-Wesley, 2011).
34. Nielsen, M. A. & Chuang, I. L. *Quantum Computation and Quantum Information* (Cambridge University Press, 2000).
35. Campbell, E. T. & Howard, M. Unified framework for magic state distillation and multiqubit gate synthesis with reduced resource cost. *Phys. Rev. A* **95**, 022316 (2017).
36. Fowler, A. G., Mariantoni, M., Martinis, J. M. & Cleland, A. N. Surface codes: Towards practical large-scale quantum computation. *Phys. Rev. A* **86**, 032324 (2012).
37. Maslov, D. Advantages of using relative-phase Toffoli gates with an application to multiple control Toffoli optimization. *Phys. Rev. A* **93**, 022311 (2016).
38. Amy, M., Maslov, D., Mosca, M. & Roetteler, M. A meet-in-the-middle algorithm for fast synthesis of depth-optimal quantum circuits. *IEEE Trans. CAD Integrated Circuits Syst.* **32**, 818–830 (2013).
39. Gosset, D., Kliuchnikov, V., Mosca, M. & Russo, V. An algorithm for the *T*-count. *Quantum Inf. Comput.* **14**, 1261–1276 (2014).
40. Jones, C. Low-overhead constructions for the fault-tolerant Toffoli gate. *Phys. Rev. A* **87**, 022328 (2013).
41. Gidney, C. Halving the cost of quantum addition. *Quantum* **2**, 10–22331 (2018).
42. Selinger, P. Quantum circuits of *T*-depth one. *Phys. Rev. A* **87**, 042302 (2013).
43. Chan, S. M. *Pebble games and complexity*. Ph.D. thesis, University of California, Berkeley (2013).
44. Chan, S. M., Lauria, M., Nordstrom, J. & Vinyals, M. Hardness of approximation in PSPACE and separation results for pebble games. In *2015 IEEE 56th Annual Symposium on Foundations of Computer Science*, 466–485 (2015).
45. Knill, E. An analysis of Bennett's pebble game. Preprint at https://arxiv.org/abs/math/9508218 (1995).
46. Soeken, M. et al. The EPFL logic synthesis libraries. Preprint at https://arxiv.org/abs/1805.05121 (2018).

ACKNOWLEDGEMENTS

This research was supported by the Swiss National Science Foundation (200021-169084 MAJesty).

AUTHOR CONTRIBUTIONS

G.M. and M.S. conceived the algorithms and planned the experimental evaluation. G.M. implemented the algorithms, performed the experiments and analyzed the data. G.D.M. coordinated the project. G.M. wrote the manuscript. All authors revised and approved the content of the manuscript.

COMPETING INTERESTS

The authors declare no competing interests.

ADDITIONAL INFORMATION

Correspondence and requests for materials should be addressed to Giulia Meuli.

Reprints and permission information is available at http://www.nature.com/reprints

Publisher's note Springer Nature remains neutral with regard to jurisdictional claims in published maps and institutional affiliations.