Reversible Pebble Games For Reducing Qubits In Hierarchical Quantum Circuit Synthesis

Debjyoti Bhattacharjee*, Mathias Soeken†, Srijit Dutta‡, Anupam Chattopadhyay* and Giovanni De Micheli†
*School of Computer Science and Engineering, Nanyang Technological University, Singapore - 639798
†Integrated Systems Laboratory, EPF Lausanne, Switzerland. ‡Indian Institute of Technology Bombay, India
Email: *debjyoti001@ntu.edu.sg*

***Abstract*—Hierarchical reversible logic synthesis can find quantum circuits for large combinational functions. The price for a better scalability compared to functional synthesis approaches is the requirement for many additional qubits to store temporary results of the hierarchical input representation. However, implementing a quantum circuit with large number of qubits is a major hurdle. In this paper, we demonstrate and establish how reversible pebble games can be used to reduce the number of stored temporary results, thereby reducing the qubit count. Our proposed algorithm can be constrained with number of qubits, which is aimed to meet. Experimental studies show that the qubit count can be significantly reduced (by up to 63.2%) compared to the state-of-the-art algorithms, at the cost of additional gate count.**

I. Introduction

The race for establishing quantum supremacy [1] is at its height right now. Researchers across the world are identifying applications, which can achieve at least super-polynomial speed-up for a quantum computer over the best possible classical one. In parallel, by extending the limits of classical computing systems, up to 56 qubit quantum circuit is recently simulated on a supercomputer [2]. Though this limit is surpassed by the recent quantum processor from Google with 72 qubits [3], yet it is not sufficient to establish quantum supremacy due to the error that builds up during the quantum computation [4]. To tackle this challenge, while on the one hand, it is imperative to physically implement multi-qubit gates with low error rates; on the other hand, it is important to map quantum algorithms to a compact circuit for achieving the quantum supremacy with as low resources as possible.

The challenge of synthesizing an algorithm on to a set of quantum gates gave rise to the domain of quantum/reversible logic synthesis, with a growing body of techniques that achieve excellent scalability, efficient circuit construction in terms of T-count, T-depth and qubit count. Hierarchical reversible logic synthesis (LHRS) represent the state-of-the-art synthesis technique for large combinational functions [5], [6]. While this technique present a good balance between the scalability, T-count and T-depth, it does not particularly emphasize the optimization of qubit count, which, we argue as the dominant problem towards the practical realisation of the first batch of quantum algorithms. This is the main focus of our work.

Optimisation of qubit count is a well-studied problem. First, there have been numerous studies on minimising the ancilla qubits [7], leading towards the ancilla-free, scalable synthesis approaches [8]. Second, by taking cue from Bennett's reversible pebble game[1] [10], [11], several heuristics have been proposed to include the "uncompute" stage [12], [13], [14]. However, none of these works presented a pebble game heuristic integrated with a quantum logic synthesis flow, which is also observed here [15]. The work closest to ours is by Parent et al. [12] that implemented pebbling strategies and games to demonstrate trade-off between qubit count and circuit depth. This work, however, does not explicitly address the problem of circuit synthesis. Further, the pebbling strategy needs to be closely integrated with the overall synthesis technique, which we undertook here. Our contributions are noted as following.

- A synthesis algorithm to map from LUT network to single target gates, using reversible pebble game, to reduce number of qubits has been proposed.
- Two optimisations have been proposed to lower the number of single target gates in the synthesized circuit.
- A thorough benchmarking for large combinational circuits that validate the presented heuristics has been undertaken. Compared to the state-of-the-art techniques, we could significantly reduce the qubit count.

II. Preliminaries

A. Boolean Logic Network

A Boolean logic network is a DAG, with inputs, outputs and gates represented as vertices while edges connect gates to the inputs, outputs and other gates. A Boolean logic network N can be formally defined as a three tuple, $N = \langle V, E, F\rangle$ where $\langle V, E\rangle$ is a DAG and F is the mapping function. The vertex set V is the union of primary inputs P, primary outputs O and gates G, $V = P \cup O \cup G$. The edge set E is a set of ordered tuples (v_i, v_j), such that $i \neq j$ and $v_i, v_j \in V$. Corresponding to each vertex, the in-degree $\delta^-(v)$ and out-degree $\delta^+(v)$ of a vertex v can be defined as $\delta^-(v) = |(v, v')|$ and $\delta^+(v) = |(v'', v)|$ where $\{v, v', v''\} \in V$ and $\{(v, v'), (v'', v)\} \in E$. Each gate $g \in G$ represents a Boolean function $F(g) : \mathbb{B}^{\delta^-(g)} \rightarrow \mathbb{B}$. The *cone* of a vertex v is the set of all vertices in G, that have a path to the vertex v. A Boolean network N is termed as k-feasible, if $\delta^-(v) \leq k, \forall v \in V$. Such k-feasible networks

[1]The irreversibility-space trade-off in the pebble game is also useful in other domains, such as, register allocation stage of compiler [9].

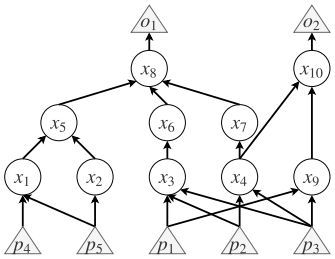

Fig. 1: *A 3-feasible network with five inputs and two outputs.*

are also referred as k-LUT network (LUT stands for Look-Up Table). Homogeneous Boolean networks such as And-Inverter Graph [16] and Majority-Inverter Graphs [17], can be transformed into k-feasible networks by means of LUT mapping algorithms [18], [19].

Example 1. *Figure 1 shows a 3-feasible network. The vertex set is* $V = P \cup O \cup G$ *where* $P = \{p_1, p_2, \ldots, p_5\}$, $O = \{o_1, o_2\}$ *and* $G = \{x_1, x_2, \ldots, x_{10}\}$. $cone(o_1) = \{x_1, x_2, \ldots, x_8\}$. $cone(o_2) = \{x_4, x_9, x_{10}\}$

B. Reversible Logic Network

A reversible logic network realizes a reversible function by means of a cascade of reversible gates. In this paper, we consider generic *single target gate* (STG). A STG T_c $(\{x_1, x_2, \ldots, x_k\}, x_{k+1})$ has k control lines $x_1, \ldots, x_k$, a single target line x_{k+1} and a control function $c : \mathbb{B}^{k+1} \rightarrow \mathbb{B}$. The gate realizes a reversible function $f : \mathbb{B}^{k+1} \rightarrow \mathbb{B}^{k+1}$ with $x_i \longmapsto x_i$, $\forall i \leq k$ and $x_{k+1} \longmapsto x_{k+1} \oplus c(x_1, \ldots, x_k)$.

C. Mapping to Quantum Circuit

An elementary method to map a LUT network to reversible circuit is by means of using a STG for each LUT in the topological ordering. The target for the STG is chosen to be an ancilla line initialized to '0'. The circuit should not have any garbage lines to allow implementation on a quantum computer. This constraint arises due to the fact that result of the calculation is entangled with the intermediate results and thus they cannot be discarded and reused without damaging the results they are entangled with [20]. To disentangle the qubits and revert the targets to their initial state (constant 0), the STGs for the LUTs computing the intermediate results should be applied in the reverse topological order.

A STG can be computed on a free qubit (constant 0), if all its predecessors have been computed. Also, the qubit with the result of the STG can be returned to state 0, only when all the predecessors are in computed state. This is the primary constraint on *compute* and *uncompute* of STG. We express the possible operations using the following notations:

- $\texttt{PI}(x, q)$ for primary input $x \in P$ and qubit q: This assigns input x to qubit q in the circuit.
- $\texttt{COMP}(g, q)$ for gate $g \in G$ and qubit q: This applies a single target gate $T_{F(g)}(m(j)|j \in \delta^-, q)$ and sets $m(g) \leftarrow q$.
- $\texttt{UCOMP}(g, q)$ for gate $g \in G$ and qubit q: This acts same as $\texttt{COMP}(g, q)$ but sets $m(g) \leftarrow 0$. We represent this using g^{ψ} in the circuit diagrams.
- $\texttt{PO}(x, q)$ for primary output $x \in O$ and qubit q: This assigns output x to qubit q in the circuit.

Example 2. *The second dashed box in Figure 2 shows the circuit of mapping the cone of output* o_2 *using STGs. Each of the primary input is mapped to a unique qubit. Nodes* x_4 *and* x_9 *are computed first. This is followed by computing the node* x_{10}*, which is the output* o_2*. Then the intermediate nodes are uncomputed (represented by* x_i^{ψ}*). The entire circuit can be expressed in terms of operations as follows :* $\texttt{PI}(p_1, q_1)$, $\texttt{PI}(p_2, q_2)$, $\texttt{PI}(p_3, q_3)$, $\texttt{COMP}(x_4, q_4)$, $\texttt{COMP}(x_9, q_5)$, $\texttt{COMP}(x_{10}, q_6)$, $\texttt{UCOMP}(x_9, q_5)$, $\texttt{UCOMP}(x_4, q_4)$, $\texttt{PO}(o_2, q_6)$.

A STG $(T_c(\{x_1, x_2, \ldots, x_k\}, x_{k+1}))$ can be decomposed into a cascade of multi-control Toffoli gates (a special STG with control function as either 1,i.e., tautology or a single product term).

$$T_{c1}(X_1, x_{k+1}) \circ T_{c1}(X_1, x_{k+1}) \circ \ldots \circ T_{c1}(X_1, x_{k+1})$$

Individual Toffoli gates can be mapped to the fault tolerant Clifford+T gate library for native implementation of a quantum computer. The Clifford+T gate library consists of the reversible CNOT gates, the Hadamard gate and the T-gate. Since the T-gate has considerable higher implementation cost compared to the other gates, the cost of the other gates are customarily neglected [21]. Multiple techniques exist in literature for mapping multi-control Toffoli gates into Clifford+T circuits [22].

III. Qubit count optimization

This section presents a method to derive the upper bound on the number of qubits required for mapping a LUT network. An introduction to reversible pebble games, followed by a detailed example is presented. We then proceed to explain the proposed heuristic for reversible pebble games and two effective optimizations for the same.

A. Upper bound on the number of qubits

Given a LUT network with multiple outputs, the cone of each output can be considered separately as an LUT network. For each of the output, the output cone can be computed, followed by immediately uncomputing the intermediate results in the cone. This approach can be used to obtain an upper bound N_{naive} on the number of qubits required for computing a LUT network.

Lemma III.1. *A LUT graph with* N_i *inputs,* N_o *outputs and* $N_c = max(cone(n_i))$*, for* $1 \leq i \leq m$*, can be computed with at most* $N_{naive} = N_i + N_o + N_c - 1$ *qubits.*

Example 3. *For the DAG shown in Figure 1,* $N_i = 5$*,* $N_o = 2$ *and* $N_c = max(8, 3) = 8$*. Therefore, the upper bound on the number of qubits required to map the graph is* $N_{naive} = 14$*.*

B. Reversible Pebble Game

Bennett introduced the reversible pebble games [11] in the context of reversible computation, which is the exact problem we are addressing in this paper. Given a DAG G with a

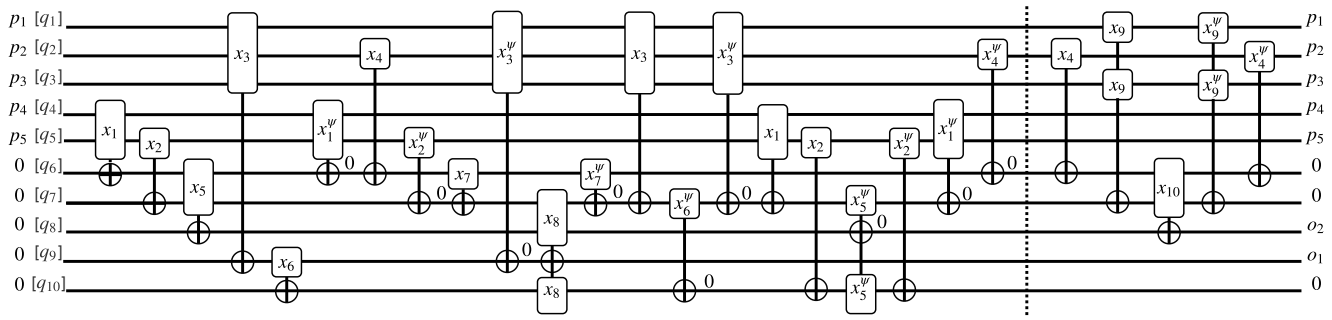

Fig. 2: *Mapping of the DAG shown in the Figure 1, using reversible pebble game. The rectangles denote the input lines of the STG, while the* $\oplus$ *denotes the target line of the STG.*

unique sink node s, the reversible pebble games starts with no pebbles on G and terminates with a pebble (*only*) on the sink s. Placing and removal of pebble on a node is governed by the following two rules: *(1) To pebble a node v, all the predecessors of v must be pebbled. (2) To unpebble a node v, all the predecessors of v must be pebbled.* In the context of computing a LUT network, the LUT network is the DAG while the pebbles represents qubits. Pebbling a node is equivalent to *computing* a node while either "unpebbling a node" or "unpebble operation" is same as *uncompute* operation of a node. The minimum number of pebbles required to pebble a DAG using reversible pebble game by Bennett is termed as the *reversible pebble number* of the graph. Determining the reversible number of a given DAG with a unique sink node (single output) is PSPACE-complete [23].

Example 4. *Figure 2 shows the circuit mapped using reversible pebble games for the DAG in Figure 1. We assume that* 10 *qubits are available for mapping.* 5 *of the qubits are used for mapping each of the primary inputs $p_1, \ldots, p_5$. We consider mapping the output cone o_1 first, followed by o_2. Nodes x_1, x_2 are computed, followed by computing their successor x_5. Then node x_3 is computed, followed by x_6. At this point, none of the qubits are free for mapping rest of the nodes. We uncompute x_1 (represented by x_1^{ψ} in the circuit) and use the free qubit to compute x_4. Similarly, x_2 is uncomputed and x_7 is computed on that qubit. The qubit for x_3 is now uncomputed to map x_8 (which drives the output o_2).*

Now, we begin freeing the allocated qubits in this cone. Node x_7 is uncomputed. Since the predecessor x_3 of node x_6 is not in computed state, x_6 cannot be uncomputed. This is due to rule (2) of pebble game. Therefore, x_3 is computed. This is followed by uncompute of x_6 and x_3. For uncomputing x_5, predecessors x_1 and x_2 are computed. Then, x_5, x_1 and x_2 are uncomputed. Finally x_4 is uncomputed. This completes computation of cone o_1.

Computation of cone o_2 is trivial since there are only 3 qubits to map and 4 qubits are available for mapping. Predecessors x_4 and x_9 are computed, followed by compute of node x_{10}. Then, intermediate nodes x_9 and x_4 are uncomputed.

C. Reversible Pebble Game Heuristic

The procedure takes a LUT-network $N = \langle V, E, F \rangle$ and number of available qubits Q_A as input. The procedure returns the number of qubits required Q_R and the corresponding sequence of operations S to realize the LUT network. Since the procedure is a heuristic, Q_R can be greater that Q_A if the heuristic cannot find a feasible sequence using Q_A available qubits.

Algorithm 1 presents the reversible pebble game heuristic in detail. The output cones are ordered in decreasing order of the size of their cones. Each cone is computed, followed by immediate uncompute of all the nodes, except the output node for that cone. The ELIGIBLE procedure returns $True$ for a node, if all the successors of that node are in computed state.

TABLE I: *Sub-functions of reversible pebble game heuristic.*

Function	Definition
$level(n)$	$\begin{cases} 0, n \in P \\ max_{n_i \in \delta^-(n)}(level(n_i)+1), n \notin P \end{cases}$
$pending(n)$	$\lvert n_i \rvert$, $n_i \notin computedV$, $n_i \in \delta^-(n)$
$contrib(n)$	$\sum_{n_o \in \delta^+(n)} \frac{1}{pending(n_0)}$
$succDep(n) = True$	$n_o \in computed$, $\forall n_o \in \delta^+(n)$
$succUComp(n) = True$	$n_o \notin computedV$, $\forall n_o \in \delta^+(n)$

Algorithm 2 describes the computation of an output cone. Some of the functions used are summarily defined in Table I. Initially, the nodes ($fanIn$) in the cone with only primary inputs as predecessors are eligible for computation. We determine the order of computation between eligible nodes as follows. If a node in a level l in the graph is eligible for compute, then it must be computed before any node at a lower level $l-1$. If there are multiple such eligible nodes, then we choose the node that would contribute most towards making its successor eligible for computing. We retrieve the top node for the priority queue, get a qubit to compute this node on, map this node's computation to the qubit, update the priority queue and finally add all the uncomputed successors of this node to the priority queue.

One of the key challenges is to find a free qubit (constant 0) for performing the computation, which is defined in the GETQUBIT procedure. If there is an available qubit (stored in stack A), then it can be used. Otherwise, we search

Algorithm 1 Reversible Pebble Game Heuristic

```
1: procedure PEBBLE(N,Q_A)
   ▷ Logic Network N = ⟨V = (P ∪ O ∪ G), E, F⟩
   ▷ Q_A=Available qubits
2:   Initialize empty stack A.
3:   for i in range(1, Q_A) do
4:     A.push(i);
5:   end for
6:   Q_R = Q_A
7:   Initialize empty queue S
8:   for n_i ∈ PI do
9:     q = GETQUBIT(n_i,PI);
10:    S.push_back(PI(n_i,q))
11:  end for
12:  Initialize empty set ComputedV
13:  Sort O in descending order of cone(n_o), for n_o ∈ O
14:  for n_o ∈ PO do
15:    computed = COMPUTEOUTPUT(n_o);
16:    UNCOMPUTEOUTPUT(n_o, computed);
17:    S.push_back(PO(n_o,map[n_o]));
18:  end for
   ▷ Synthesis sequence S, number of required qubits Q_R
19:  return S, Q_R;
20: end procedure
21: procedure ELIGIBLE(node)
22:  if n ∈ computed, ∀n ∈ δ⁻(node) then
23:    return True;
24:  else
25:    return False;
26:  end if
27: end procedure
28: procedure GETQUBIT(out,type)
29:  if !A.empty() then
30:    q = A.pop();
31:    return q
32:  end if
   ▷ Iterate over eligible nodes in topological order (higher level considered
   first)
33:  l = max_{n∈cone(out)}(level(n));
34:  while l > 0 do
35:    N = None
36:    E = {n} | ELIGIBLE(n), level(n)=l ,n ∉ predDep
37:    if !E.empty() then
38:      if type == COMP then
39:        if ∃n ∈ E | succUComp(n), succDep(n) then
40:          N = n;
41:        else if ∃n ∈ E | succDep(n) then
42:          N = n;
43:        end if
44:      else if type == UCOMP then
45:        N = n|n ∈ E;
46:      end if
47:      if N != None then
48:        q = map[N]
49:        computedV.remove(n)
50:        S.push_back(UCOMP(N,q))
51:        return q
52:      end if
53:    end if
54:    l = l − 1;
55:  end while
   ▷ No eligible nodes present with all successors computed at least once.
56:  Q_R = Q_R + 1;                          ▷ New qubit allocated
57:  return Q_R;
58: end procedure
```

topologically (from higher to lower) for eligible qubits which are not part of the current dependency ($predDep$). During compute, these include just the predecessors of the node under consideration. Out of these eligible nodes, we should not uncompute a node for which a successor has not been computed even once (defined as $succDep$ function) — this guarantees progress of the algorithm. Moreover, it is intuitive to choose a node with all its successors uncomputed (specified by $succUComp$ function) over a node that still has computed successors, since this increases the chances of more eligible

Algorithm 2 Computing an output cone

```
59: procedure COMPUTEOUTPUT(out)
60:   fanIn = {n} such that n_f ∈ P, ∀n_f ∈ δ⁻(n) and n ∈ cone(out)
61:   pendingQ = prioQ()
62:   for n ∈ fanIn do
63:     priority = COMPUTEPRIORITY(n)
64:     pendingQ.insert((n,priority))
65:   end for
66:   computed = list()
67:   while out ∉ computed do
68:     node = pendingQ.pop()
69:     Initialize empty set predDep
70:     for n_i ∈ δ⁻(node) do
71:       predDep.insert(n_i);
72:     end for
73:     q = GETQUBIT(out,COMP)
74:     map[node] = q
75:     computed.add(node)
76:     computedV.add(node)
77:     S.push_back(COMP(node,qubit))
78:     UPDATEQUEUE(pendingQ);
79:     for n_o ∈ δ⁺(n) do
80:       if n_o ∉ computed and ELIGIBLE(n_o) then
81:         pendingQ.insert((n_o, COMPUTEPRIORITY(n_o)))
82:       end if
83:     end for
84:   end while
85:   return computed
86: end procedure
87: procedure COMPUTEPRIORITY(node)
   ▷ Higher level/contribution of node indicates higher priority.
88:   return (level(node), contrib(node))
89: end procedure
```

nodes to be freed for reuse in the future. Finally, if a node N eligible for uncompute is found, the qubit is freed, the node is removed from the computed node set and an uncompute step is added to the sequence S. However, it can happen that none of the computed nodes are eligible to be uncomputed. In that case, a new qubit is added.

The order of computation of a node for the first time, stored in the list $computed$, that is the input parameter to the UNCOMPUTEOUTPUT procedure. The nodes are uncomputed in the reverse order of compute. If the node is already uncomputed, then nothing needs to be done. However, if it is computed but not eligible, then its predecessors have to be computed first. Then, the node is uncomputed and the frees qubit q is added to free qubit stack A. Also, the node is removed from the computed node set $computedV$.

D. Optimizations

We present two optimizations based on the outputs returned by the heuristic — number of qubits required Q_R and the sequence of the operations S.

1) Iterate Heuristic O_1: The number of required qubits Q_R can be higher than that the number of available qubits Q_A. When there are no free qubits, the heuristic aggressively tries to unpebble the graph. This leads to additional COMP/UCOMP operations. Specifically, let us consider the case when $Q_A << Q_R < N_{naive}$. Some of these steps can be avoided by running a second round of the heuristic with $Q_A = Q_R$ as input. This results in considerable reduction in the number of operations, without any significant increase in number of qubits.

2) Optimize Sequence O_2: The output of the sequence of step S can be optimized by eliminating redundant operations. Let $S[i]$ denote the i^{th} operation in the sequence. Two oper-

Algorithm 3 Uncomputing an output cone.

```
90: procedure UNCOMPUTEOUTPUT(out, computed)
    ▷ Store reversed computed list in rComputed, except the output node
91:   Initialize an empty list rComputed
92:   for i ∈ range(1, len(computed) − 1) do
93:     rComputed.push_back(computed[len(computed) − i − 2])
94:   end for
95:   while !rComputed.empty() do
96:     n = rComputed[1];
97:     rComputed.delete(1);
98:     if n ∉ computedV then
99:       continue;
100:    else if !ELIGIBLE(n) then
101:      Initialize empty set predDep
102:      COMPUTEPRED(n);
103:    end if
104:    q = map[n]
105:    A.push(q)
106:    S.push_back(UCOMP(n,q))
107:    computedV.remove(n)
108:  end while
109: end procedure
104: procedure COMPUTEPRED(n)
105:  for n_i ∈ δ⁻(n) do
106:    predDep.insert(n_i);
107:  end for
108:  for n_i ∈ δ⁻1(n)) do
109:    if n_i ∈ computedV then
110:      continue;
111:    else if ! ELIGIBLE(n_i) then
112:      COMPUTEPRED(n_i)
113:    end if
114:    q = GETQUBIT(n_i,UCOMP)
115:    map[n_i] = q
116:    S.push_back(COMP(n_i,q))
117:    computedV.insert(n);
118:  end for
119: end procedure
```

ations $S[i]$ and $S[i+1]$ on a node $n|n \in G$ are redundant iff

- $S[i]$ =COMP(n,q) , $S[i+1]$ =UCOMP(n,q)
- $S[i]$ =UCOMP(n,q) , $S[i+1]$ =COMP(n,q)

The redundant operations are repeatedly eliminated, till no further elimination is feasible. The LUT networks enforce sharing of logic to minimize the number of nodes in the network, which results in plenty of overlap between cones of the individual outputs. Also, the priority of nodes for computation remains similar, across output cones. Thus, the uncomputation order of multiple nodes for a cone is exactly reverse of the order of computation for these nodes in a different cone, with these nodes common to the cones. This enables the effectiveness of the optimization O_2.

IV. Experimental results and discussion

We have implemented Reversible Pebble Game Heuristic (RPGH) as part of the *lhrs* command in the reversible logic synthesis framework RevKit [24]. The experiments were performed on the EPFL benchmarks[2], along with their best LUT mapping results (_bl suffix). *lhrs* first derives a LUT network from an And Inverter graph (AIG) which is then mapped into a circuit of STGs using RPGH. We set the number of available qubits $Q_A = 0.8(max_{n_v \in V}(cone(n_v)) + N_o) + N_i)$ for each benchmark and use it as input to RPGH. The synthesis results of RPGH are presented in Table II. #LUT denotes the number of LUTs for the given cut size (k). $\mathbf{Q_R}$ and #STG denote number of qubits and single target gates in the synthesized circuit obtained using RPGH respectively. We compare our results with the existing implementation results ($\#Q_{orig}$, $\#STG_{orig}$) of LHRS, available in RevKit. The RPGH algorithm manages to reduce (denoted by $\Delta\%$) the number of qubits for all the benchmarks, which is the primary goal of this paper. Naturally, the number of single-target gates (denoted by α) significantly increases using the proposed algorithm, that leads to increased quantum costs (see, e.g., [25]).

To obtain a Clifford+T quantum circuit from the STG mapped circuit, each single-target gate has to be mapped into a Clifford+T quantum circuit. The RPGH algorithm is compatible to all varieties of STG to Clifford+T mapping algorithms (e.g., [26]). As a representative example, we show the results of mapping the synthesized STG circuit to Clifford+T circuit using various algorithms for the *sin* benchmark in Table III. The T-count of the synthesized circuits are reported, along with the qubit and STG count. Any improvement in the mapping algorithm of single-target gates to quantum circuits would directly benefit our algorithm, for reducing T-count. For $k = 6$ as an example, we can see that using the *direct+def* mapping script results in lower T-count than *mindb+def_wo4* script while having the same qubit count.

By partitioning the logic network into LUTs with a greater number of inputs (k), the number of LUTs in the LUT network is lesser. Since the RPGH algorithm uses the LUT network as input and each LUT gets mapped to an STG, a smaller network helps RPGH to map the network using smaller number of qubits, with lower number of STGs. However, since a k-input LUT would map to a k-input STG, using a greater cut size results in an STG with greater number of inputs and thus has a higher quantum cost when mapped using the Clifford+T library.

Finally, we demonstrate the space (Q_R) and time (T-count) trade-off feasible by using RPGH in Table IV. By varying φ from 0.3 to 0.8, we vary the input number of qubits Q_A to RPGH. The RPGH algorithm maps the circuit with Q_R qubits, which is close to the input Q_A. This demonstrates the effectiveness of the algorithm in using reversible pebbles game to meet input qubit constraint. Using a lower number of qubits for mapping leads to a higher T-count and vice-versa. Similar results are observed for other benchmarks as well, but not reported due to lack of space. To the best of our knowledge, the proposed RPGH algorithm is the first one to permit a constraint on number of qubits as an input for quantum logic synthesis.

V. Conclusion

Implementation of scalable quantum circuits is among the most significant scientific challenges of current times. Existing design automation flows for quantum circuits tend to emphasize on the number of gates and logical depth. In contrast, we draw attention to the reversible pebble game, which presents an opportunity to reduce the qubits. In this paper, for the first time, we present a heuristic for lowering the qubits integrated within a scalable, hierarchical logic synthesis flow.

[2]https://lsi.epfl.ch/benchmarks

TABLE II: *Synthesis results on the EPFL arithmetic benchmarks.* $\Delta\% = \frac{(\#Q_{orig} - \#Q) * 100}{\#Q_{orig}}$, $\alpha = \frac{\#STG}{\#STG_{orig}}$.

		k=6			k=10			k=16		
Benchmark	$\mathbf{N_i/N_o}$	**#LUT**	$\mathbf{Q_R}$ ($\Delta\%$)	**#STG**(α)	**#LUT**	$\mathbf{Q_R}$ ($\Delta\%$)	**#STG**(α)	**#LUT**	$\mathbf{Q_R}$ ($\Delta\%$)	**#STG**(α)
adder	256/129	249	386 (23.6)	14916 (40.4)	234	386 (21.2)	11608 (34.2)	207	386 (16.6)	6823 (23.9)
adder_bl	256/129	192	386 (13.8)	8190 (32.1)	189	386 (13.3)	7688 (30.9)	187	386 (12.9)	7206 (29.4)
bar	135/128	512	276 (52.7)	11932 (13.3)	512	276 (52.7)	11932 (13.3)	510	275 (52.7)	11736 (13.2)
bar_bl	135/128	768	309 (63.2)	16782 (11.9)	668	286 (61.4)	12056 (10)	523	271 (54.5)	5788 (6.3)
div	128/128	23863	10273 (17.1)	2409454 (50.6)	23321	9954 (17.4)	2345136 (50.4)	22910	9691 (18.1)	2295414 (50.2)
div_bl	128/128	3271	3087 (9.2)	634450 (98.9)	3098	3091 (4.2)	597723 (98.5)	2889	2604 (13.7)	548925 (97.2)
log2	32/32	7579	6100 (19.9)	523327 (34.6)	2843	2347 (18.4)	185672 (32.8)	2283	1925 (16.8)	143166 (31.6)
log2_bl	32/32	6595	5309 (19.9)	460563 (35)	3006	2440 (19.7)	203898 (34.1)	2022	1670 (18.7)	136675 (34.1)
max	512/130	721	1092 (11.4)	177191 (135.1)	389	825 (8.4)	73812 (113.9)	284	746 (6.3)	44152 (100.8)
max_bl	512/130	524	934 (9.8)	113126 (123.2)	328	777 (7.5)	55765 (106)	213	695 (4.1)	23467 (79.3)
multiplier	128/128	5678	4678 (19.4)	770713 (68.6)	2977	2515 (19)	378558 (65)	2724	2315 (18.8)	343227 (64.5)
multiplier_bl	128/128	4923	4068 (19.4)	666365 (68.6)	3291	2766 (19.1)	428239 (66.4)	2426	2079 (18.5)	306224 (64.8)
sin	24/25	1444	1181 (19.6)	73093 (25.5)	690	580 (18.8)	31165 (23)	494	433 (16.4)	21330 (22.1)
sin_bl	24/25	1262	1028 (19.5)	63825 (25.5)	534	453 (18.8)	24563 (23.6)	391	339 (18.3)	17511 (23.1)
sqrt	128/64	8084	6647 (19.1)	367572 (22.8)	7764	6391 (19)	346197 (22.4)	7688	6330 (19)	337900 (22.1)
sqrt_bl	128/64	3076	2640 (17.6)	138108 (22.7)	2746	2377 (17.3)	120951 (22.3)	2500	2181 (17)	106640 (21.6)
square	64/128	3992	3266 (19.5)	526175 (67)	3289	2714 (19.1)	424555 (65.8)	2598	2166 (18.7)	331977 (65.5)
square_bl	64/128	3244	2669 (19.3)	422177 (66.4)	2816	2333 (19)	360257 (65.4)	2232	1877 (18.3)	282087 (65)

TABLE III: *T-count of synthesized circuit using various STG to Clifford+T mapping algorithms [26] for* sin *benchmark.*

		Mapping	**direct**		**mindb**	
		Script	def	def_wo4	def	def_wo4
k	$\mathbf{Q_R}$	**#STG**	**T-count**	**T-count**	**T-count**	**T-count**
6	1181	73093	2331631	2318301	2927440	2850017
10	580	31165	9091604	9123544	14631353	14632203
16	433	21330	93594652	94909560	43922561	43949572

TABLE IV: *Qubit count (Q_R) and T-count of synthesized circuit for varying number of available qubits [$Q_A = 0.8(max_{n_v \in V}(cone(n_v)) + N_o) + N_i$)=$\varphi(1421 + 25) + 24$] for* sin *benchmark with cut size $k = 6$.*

φ	**0.3**	**0.4**	**0.5**	**0.6**	**0.7**	**0.8**
$\mathbf{Q_A}$	458	603	747	892	1037	1181
$\mathbf{Q_R}$	590	604	747	896	1037	1181
T-count	3415534	3413100	3288854	3147188	2634309	2331631

This reduces the number of qubits by up to 62.3% compared to the baseline, state-of-the-art synthesis techniques.

REFERENCES

[1] J. Preskill, "Quantum computing and the entanglement frontier," *arXiv preprint arXiv:1203.5813*, 2012.

[2] E. Pednault, J. A. Gunnels, G. Nannicini, L. Horesh, T. Magerlein, E. Solomonik, and R. Wisnieff, "Breaking the 49-qubit barrier in the simulation of quantum circuits," *arXiv preprint arXiv:1710.05867*, 2017.

[3] "Bristlecone, google's new quantum processor," https://research.googleblog.com/2018/03/a-preview-of-bristlecone-googles-new.html, accessed: 2018-04-21.

[4] M. Dyakonov, "Is fault-tolerant quantum computation really possible?" *arXiv preprint quant-ph/0610117*, 2006.

[5] M. Soeken, M. Roetteler, N. Wiebe, and G. De Micheli, "Hierarchical reversible logic synthesis using LUTs," in *DAC*. IEEE, 2017, pp. 1–6.

[6] M. Soeken and A. Chattopadhyay, "Unlocking efficiency and scalability of reversible logic synthesis using conventional logic synthesis," in *Proceedings of the 53rd Annual Design Automation Conference*. ACM, 2016, p. 149.

[7] M. M. Rahman, G. W. Dueck, A. Chattopadhyay, and R. Wille, "Integrated synthesis of linear nearest neighbor ancilla-free mct circuits," in *Multiple-Valued Logic (ISMVL), 2016 IEEE 46th International Symposium on*. IEEE, 2016, pp. 144–149.

[8] M. Soeken, L. Tague, G. W. Dueck, and R. Drechsler, "Ancilla-free synthesis of large reversible functions using binary decision diagrams," *Journal of Symbolic Computation*, vol. 73, pp. 1–26, 2016.

[9] R. Sethi, "Complete register allocation problems," *SIAM journal on Computing*, vol. 4, no. 3, pp. 226–248, 1975.

[10] C. H. Bennett, "Time/space trade-offs for reversible computation," *SIAM Journal on Computing*, vol. 18, no. 4, pp. 766–776, 1989.

[11] C. H. Bennett, "Logical reversibility of computation," *IBM journal of Research and Development*, vol. 17, no. 6, pp. 525–532, 1973.

[12] A. Parent, M. Roetteler, and K. M. Svore, "Reversible circuit compilation with space constraints," *arXiv preprint arXiv:1510.00377*, 2015.

[13] M. Amy, "Algorithms for the optimization of quantum circuits," Master's thesis, University of Waterloo, 2013.

[14] A. Shafaei, M. Saeedi, and M. Pedram, "Reversible logic synthesis of k-input, m-output lookup tables," in *Proceedings of the Conference on Design, Automation and Test in Europe*. EDA Consortium, 2013, pp. 1235–1240.

[15] M. Soeken, M. Roetteler, N. Wiebe, and G. De Micheli, "Logic synthesis for quantum computing," *arXiv preprint arXiv:1706.02721*, 2017.

[16] A. Kuehlmann, V. Paruthi, F. Krohm, and M. K. Ganai, "Robust boolean reasoning for equivalence checking and functional property verification," *IEEE TCAD*, vol. 21, no. 12, pp. 1377–1394, 2002.

[17] L. Amaru, P.-E. Gaillardon, and G. De Micheli, "Majority-inverter graph: A new paradigm for logic optimization," *IEEE TCAD*, vol. 35, no. 5, pp. 806–819, 2016.

[18] S. Ray, A. Mishchenko, N. Een, R. Brayton, S. Jang, and C. Chen, "Mapping into LUT structures," in *Proceedings of the Conference on DATE*. EDA Consortium, 2012, pp. 1579–1584.

[19] D. Chen and J. Cong, "Daomap: A depth-optimal area optimization mapping algorithm for fpga designs," in *Proceedings of the 2004 ICCAD*. IEEE Computer Society, 2004, pp. 752–759.

[20] M. A. Nielsen and I. L. Chuang, "Quantum computation and quantum information," 2000.

[21] M. Amy, D. Maslov, M. Mosca, and M. Roetteler, "A meet-in-the-middle algorithm for fast synthesis of depth-optimal quantum circuits," *IEEE TCAD*, vol. 32, no. 6, pp. 818–830, 2013.

[22] D. Maslov, "Advantages of using relative-phase Toffoli gates with an application to multiple control Toffoli optimization," *Physical Review A*, vol. 93, no. 2, p. 022311, 2016.

[23] S. M. Chan, "Just a pebble game," in *Computational Complexity (CCC), 2013 IEEE Conference on*. IEEE, 2013, pp. 133–143.

[24] "Revkit," https://msoeken.github.io/revkit.html, accessed: 2018-04-21.

[25] R. Wille, M. Soeken, D. M. Miller, and R. Drechsler, "Trading off circuit lines and gate costs in the synthesis of reversible logic," *Integration*, vol. 47, no. 2, pp. 284–294, 2014.

[26] G. Meuli, M. Soeken, M. Roetteler, N. Wiebe, and G. D. Micheli, "A best-fit mapping algorithm to facilitate ESOP-decomposition in Clifford+T quantum network synthesis," in *ASP-DAC*, 2018, pp. 664–669.